Kofax Kapow

Developer's Guide

Version: 10.3.2

Date: 2018-11-07

KOFAX

Table of Contents

Preface

Robots are executed on RoboServer through an API (Java or .Net). You may use the API directly in your own application or indirectly when you execute robots using the Management Console.

This guide consists of three parts:

- *Java Programmer's Guide*, which describes the API that can be used in Java programs.
- *.NET Programmer's Guide*, which describes the API to use in .NET applications, including C# programs.
- *Kapow Control Protocol*, which describes the creation of a JMS client for executing robots over Java Message Service (JMS), using Google Protocol Buffers (Protobuf).

Java and .NET API reference documentation is available from the API folder in your Kapow installation folder, such as `C:\Program Files\Kofax Kapow 10.3.2\API`.

Getting Help for Kofax Products

Kofax regularly updates the Kofax Support site with the latest information about Kofax products.

To access some resources, you must have a valid Support Agreement with an authorized Kofax Reseller/ Partner or with Kofax directly.

Use the tools that Kofax provides for researching and identifying issues. For example, use the Kofax Support site to search for answers about messages, keywords, and product issues. To access the Kofax Support page, go to www.kofax.com/support.

The Kofax Support page provides:
- Product information and release news

  Click a product family, select a product, and select a version number.
- Downloadable product documentation

  Click a product family, select a product, and click **Documentation**.
- Access to product knowledge bases

  Click **Knowledge Base**.
- Access to the Kofax Customer Portal (for eligible customers)

  Click **Account Management** and log in.

  To optimize your use of the portal, go to the Kofax Customer Portal login page and click the link to open the *Guide to the Kofax Support Portal*. This guide describes how to access the support site, what to do before contacting the support team, how to open a new case or view an open case, and what information to collect before opening a case.
- Access to support tools

Click **Tools** and select the tool to use.

- Information about the support commitment for Kofax products

  Click **Support Details** and select **Kofax Support Commitment**.

Use these tools to find answers to questions that you have, to learn about new functionality, and to research possible solutions to current issues.

Java Programmer's Guide

This chapter describes how to execute Robots using the Kofax Kapow Java API. The guide assumes that you have completed the Design Studio tutorials and know how to write simple robots, and that you are familiar with the Java programming language.

> **Important** The `printStackTrace` method is deprecated in Kofax Kapow version 9.6 and later.

Details about specific classes are found in the JavaDoc.

Java Basics

Robots run by the Management Console are executed using the Java API, which allows you to send requests to a `RoboServer` that instructs it to execute a particular robot. This is a classic client/server setup in which Management Console acts as the client and `RoboServer` as the server.

By using the API, any Java based application can become a client to `RoboServer`. In addition to running robots that store data in a database, you can also have the robots return data directly back to the client application. Here are some examples:

- Use multiple robots to do a federated search, which aggregates results from multiple sources in real time.
- Run a robot in response to an event on your application back end. For instance, run a robot when a new user signs up, to create accounts on web-based systems not integrated directly into your back end.

This guide introduces the core classes, and how to use them for executing robots. We will also describe how to provide input to robots, and control their execution on `RoboServer`.

The Java API is a jar file located in `/API/robosuite-java-api/lib/robosuite-api.jar` inside the Kofax Kapow installation folder. See "Important Folders" in the *Installation Guide* for details. All examples in this guide are also found in `/API/robosuite-java-api/examples`. Located next to the Java API are five additional jar files which comprise the external dependencies of the API. Most basic API tasks such as executing robots can be done without using any of these third-party libraries, while some advanced features do require the usage of one or more of these libraries. The examples in this guide specify when such libraries are required.

First Example

Let's start by looking at the code required to execute the robot named `NewsMagazine.robot`, which is located in the Tutorials folder of the default project. The robot outputs its results using the Return Value step action, which makes it easy to handle the output programmatically using the API. Other robots

(typically those run in a schedule by the Management Console) store their data directly in a database using the Store in Database step action, in which case data collected by the robot is not returned to the API client.

In the following example, we will look at how to execute the `NewsMagazine` robot and process the output programmatically.

Execute a Robot without input:

```
import com.kapowtech.robosuite.api.java.repository.construct.*;
import com.kapowtech.robosuite.api.java.rql.*;
import com.kapowtech.robosuite.api.java.rql.construct.*;

/**
 * Example that shows you how to execute NewsMagazine.robot from tutorial1
 */
public class Tutorial1 {

    public static void main(String[] args) throws ClusterAlreadyDefinedException {

        RoboServer server = new RoboServer("localhost", 50000);
        boolean ssl = false;
        Cluster cluster = new Cluster("MyCluster", new RoboServer[]{ server}, ssl);

        Request.registerCluster(cluster); // you can only register a cluster once per
application

        try {
            Request request = new Request("Library:/Tutorials/NewsMagazine.robot");
            request.setRobotLibrary(new DefaultRobotLibrary());
            RQLResult result = request.execute("MyCluster");

            for (Object o : result.getOutputObjectsByName("Post")) {
                RQLObject value = (RQLObject) o;
                String title = (String) value.get("title");
                String preview = (String) value.get("preview");
                System.out.println(title + ", " + preview);
            }
        }
    }
}
```

Let's start by looking at the classes involved and their responsibilities.

RoboServer	This is a simple value object that identifies a `RoboServer` that can execute robots. Each `RoboServer` must be activated by a Management Console and assigned KCU before use.
Cluster	A cluster is a group of RoboServers functioning as a single logical unit.
Request	This class is used to construct the robot request. Before you can execute any requests, you must register a cluster with the Request class.
DefaultRobotLibrary	A robot library instructs RoboServer on where to find the robot identified in the request. Later examples will explore the various robot library types and when/how to use them.
RQLResult	This class contains the result of a robot execution. The result contains value responses, with log and server messages.

RQLObject	Each value that is returned from a robot using the Return Value action can be accessed as an `RQLObject`.

Now let's go through each line in the example and look at the specifics.

This line tells the API that our RoboServer is running on localhost port 50000.

```
RoboServer server = new RoboServer("localhost", 50000);
```

The following code defines a cluster with a single RoboServer. The cluster is registered with the Request class, allowing you to execute requests on this cluster. Each cluster can only be registered once.

Registering a cluster:

```
boolean ssl = false;
 Cluster cluster = new Cluster("MyCluster", new RoboServer[]{ server}, ssl);
 Request.registerCluster(cluster);
```

The following code creates a request that will execute the robot named `NewsMagazine.robot` located at `Library:/Tutorials.Library:/` refers to the robot Library configured for the request. Here the `DefaultRobotLibrary` is used, which instructs `RoboServer` to look for the robot in the server's local file system. See Robot Libraries for details on how to use robot libraries.

```
Request request = new Request("Library:/Tutorials/NewsMagazine.robot");
 request.setRobotLibrary(new DefaultRobotLibrary());
```

The next line executes the robot on the cluster named `MyCluster` (the cluster previously registered) and returns the result once the robot is done. By default `execute` will throw an exception if the robot generates an API exception.

```
RQLResult result = request.execute("MyCluster")
```

Here we process the extracted values. First we get all extracted values of the type named Post and iterate through them. For each RQLObject we access the attributes of the Post type and print the result. We will look at attributes and mappings in a later section.

```
    for (Object o : result.getOutputObjectsByName("Post")) {
    RQLObject value = (RQLObject) o;
    String title = (String) value.get("title");
    String preview = (String) value.get("preview");
    System.out.println(title + ": " + preview);
 }
```

Robot Input

Most robots executed through the API are parameterized through input, such as a search keyword, or login credentials. Input to a robot is part of the request to RoboServer and is provided using the `createInputVariable` method on the request. Let us look at a short code fragment.

Input using implicit RQLObjectBuilder:

```
 Request request = new Request("Library:/Input.robot");
 request.createInputVariable("userLogin").setAttribute("username", "scott")
 .setAttribute("password", "tiger");
```

Here we create a Request and use `createInputVariable` to create an input variable named `userLogin`. We then use `setAttribute` to configure the username and password attributes of the input variable.

The preceding example is a common shorthand notation, but can also be expressed in more detail by using the RQLObjectBuilder:

Input using explicit RQLObjectBuilder:

```
Request request = new Request("Library:/Input.robot");
RQLObjectBuilder userLogin = request.createInputVariable("userLogin");
userLogin.setAttribute("username", "scott");
userLogin.setAttribute("password", "tiger");
```

The two examples are identical. The first utilizes the cascading method invocation on the anonymous `RQLObjectBuilder` and therefore shorter.

When RoboServer receives this request, the following occurs:

- RoboServer loads `Input.robot` (from whatever `RobotLibrary` is configured for the request).
- RoboServers verifies that the robot has a variable named `userLogin` and that this variable is marked as input.
- RoboServers now verifies that the attributes configured using `setAttribute` are compatible with the type of variable `userLogin`. As a result, the type must have attributes named username and password and must both be text-based attributes (the next section describes the mapping between the API and Design Studio attributes).
- If all input variables are compatible, RoboServer will start executing the robot.

If a robot requires multiple input variables, you must create all of them to execute the robot. You only have to configure required attributes; any no-required attributes that you do not configure through the API will have a null value. If you have a robot that requires a login to both Facebook and Twitter, you could define the input like this.

```
Request request = new Request("Library:/Input.robot");
request.createInputVariable("facebook").setAttribute("username", "scott")
.setAttribute("password", "facebook123");
request.createInputVariable("twitter").setAttribute("username", "scott")
.setAttribute("password", "twitter123");
```

Attribute Types

When you define a new type in Design Studio, select a type for each attribute. Some attributes can contain text, like Short text, Long Text, Password, HTML, XML, and when used inside a robot, there may be requirements to store text in these attributes. If you store text in an XML attribute, the text must be a valid XML document. This validation occurs when the type is used inside a robot, but since the API does not know anything about the type, it does not validate attribute values in the same manner. As a result, the API only has 8 attribute types versus the 19 available in Design Studio. This table shows the mapping between the API and Design Studio attribute types.

API to Design Studio mapping

API Attribute Type	Design Studio Attribute Type
Text	Short Text, Long Text, Password, HTML, XML, Properties, Language, Country, Currency, Refind Key
Integer	Integer
Boolean	Boolean
Number	Number
Character	Character
Date	Date
Session	Session
Binary	Binary, Image, PDF

The API attribute types are then mapped to Java in the following way.

Java Types for Attributes

API Attribute Type	Java Class
Text	`java.lang.String`
Integer	`java.lang.Long`
Boolean	`java.lang.Boolean`
Number	`java.lang.Double`
Character	`java.lang.Character`
Date	`java.util.Date`
Session	`com.kapowtech.robosuite.api.construct.Session`
Binary	`com.kapowtech.robosuite.api.construct.Binary`

The RQlObjectBuilder's `setAttribute` method is overloaded so you do not need to specify the attribute type explicitly when configuring an attribute through the API, as long as the right Java class is used as an argument. Here is an example that shows how to set the attributes for an object with all possible (Design Studio) attribute types.

Recommended usage of `setAttribute`**:**

```
Request request  = new Request("Library:/AllTypes.robot");
RQLObjectBuilder inputBuilder = request.createInputVariable("AllTypes");
inputBuilder.setAttribute("anInt", new Long(42L));
inputBuilder.setAttribute("aNumber", new Double(12.34));
inputBuilder.setAttribute("aBoolean", Boolean.TRUE);
inputBuilder.setAttribute("aCharacter", 'c');
inputBuilder.setAttribute("aShortText", "some text");
inputBuilder.setAttribute("aLongText", "a longer test");
inputBuilder.setAttribute("aPassword", "secret");
inputBuilder.setAttribute("aHTML", "<html>bla</html>");
inputBuilder.setAttribute("anXML", "<tag>text</tag>");
inputBuilder.setAttribute("aDate", new Date());
```

```
inputBuilder.setAttribute("aBinary", new Binary("some bytes".getBytes()));
inputBuilder.setAttribute("aPDF", (Binary) null);
inputBuilder.setAttribute("anImage", (Binary) null);
inputBuilder.setAttribute("aProperties", "name=value\nname2=value2");
inputBuilder.setAttribute("aSession", (Session) null);
inputBuilder.setAttribute("aCurrency", "USD");
inputBuilder.setAttribute("aCountry", "US");
inputBuilder.setAttribute("aLanguage", "en");
inputBuilder.setAttribute("aRefindKey", "Never use this a input");
```

The preceding example explicitly uses new Long(42L), and new Double(12.34), although 42L and 12.34 would be sufficient due to auto boxing. Also notice that we have to cast null values, because the Java compiler cannot otherwise determine which of the overloaded `setAttribute` methods to call. However, since unconfigured attributes will automatically be null, you never need to set null explicitly.

It is possible to specify the Attribute and `AttributeType` explicitly when creating input using the API. This approach is not recommended, but may be needed in rare cases, and would look like this.

Incorrect usage of `setAttribute`:

```
Request request  = new Request("Library:/AllTypes.robot");
 RQLObjectBuilder inputBuilder = request.createInputVariable("AllTypes");
 inputBuilder.setAttribute(new Attribute("anInt", "42", AttributeType.INTEGER));
 inputBuilder.setAttribute(new Attribute("aNumber", "12.34", AttributeType.NUMBER));
 inputBuilder.setAttribute(new Attribute("aBoolean", "true", AttributeType.BOOLEAN));
 inputBuilder.setAttribute(new Attribute("aCharacter", "c", AttributeType.CHARACTER));
 inputBuilder.setAttribute(new Attribute("aShortText", "some text",
 AttributeType.TEXT));
 inputBuilder.setAttribute(new Attribute("aLongText", "a longer test",
 AttributeType.TEXT));
 inputBuilder.setAttribute(new Attribute("aPassword", "secret", AttributeType.TEXT));
 inputBuilder.setAttribute(new Attribute("aHTML", "<html>bla</html>",
 AttributeType.TEXT));
 inputBuilder.setAttribute(new Attribute("anXML", "<tag>text</tag>",
 AttributeType.TEXT));
 inputBuilder.setAttribute(new Attribute("aDate", "2012-01-15 23:59:59.123",
   AttributeType.DATE));
 inputBuilder.setAttribute(new Attribute("aBinary",
   Base64Encoder.encode("some bytes".getBytes()), AttributeType.BINARY));
 inputBuilder.setAttribute(new Attribute("aPDF", null, AttributeType.BINARY));
 inputBuilder.setAttribute(new Attribute("anImage", null, AttributeType.BINARY));
 inputBuilder.setAttribute(new Attribute("aProperties", "name=value\nname2=value2",
   AttributeType.TEXT));
 inputBuilder.setAttribute(new Attribute("aSession", null, AttributeType.SESSION));
 inputBuilder.setAttribute(new Attribute("aCurrency", "USD", AttributeType.TEXT));
 inputBuilder.setAttribute(new Attribute("aCountry", "US", AttributeType.TEXT));
 inputBuilder.setAttribute(new Attribute("aLanguage", "en", AttributeType.TEXT));
 inputBuilder.setAttribute(new Attribute("aRefindKey", "Never use this a input",
   AttributeType.TEXT));
```

All attribute values must be provided in the form of strings. The string values are then converted to the appropriate Java objects based on the attribute type provided. This is only useful if you build other generic APIs on top of the Kofax Kapow Java API.

Execution Parameters

In addition to the `createInputVariable` method, the Request contains a number of methods that control how the robot executes on `RoboServer`.

Execution Control Methods on Request

`setMaxExecutionTime(int seconds)`	Controls the execution time of the robot. When this time has elapsed, the robot will be stopped by RoboServer. The timer does not start until the robot begins to execute, so if the robot is queued on RoboServer, this is not taken into account.
`setStopOnConnectionLost(boolean)`	When true (default) the robot will stop if RoboServer discovers that the connection to the client application has been lost. You should have a very good reason for setting this value to false; if your code is not written to handle this value, your application will not perform as expected.
`setStopRobotOnApiException(boolean)`	When true (default) the robot is stopped by RoboServer after the first API exception is raised. By default most steps in a Robot raise an API exception if the step fails to execute. Configure this value on the Steps Error Handling tab. When set to false, the robot continues to execute regardless of API exceptions. However, unless your application is using the `RequestExecutor` streaming execution mode, an exception is still thrown by execute(). Be extremely cautious when setting this to false.
`setUsername(String), setPassword(String)`	Sets the RoboServer credentials. RoboServer can be configured to require authentication. When this option is enabled, the client must provide credentials or RoboServer will reject the request.
`setRobotLibrary(RobotLibrary)`	A robot library instructs RoboServer on where to find the robot identified in the request. For more examples related to the various library types and their usage, see Robot Libraries.
`setExecutionId(String)`	Allows you to set the `executionId` for this request. If you don't provide one, RoboServer generates one automatically. The execution ID is used for logging and is also needed to stop the robot programmatically. The ID must be globally unique (over time). If two robots use the same execution ID, the logs will be inconsistent.
`setProject(String)`	This method is used solely for logging purposes. The Management Console uses this field to link log messages to project, so the log views can filter by project. If your application is not using the `RepositoryRobotLibrary`, you should probably set this value to inform the RoboServer logging system which project (if any) this robot belongs to.

Robot Libraries

In Design Studio, robots are grouped into projects. If you look in the file system, you will see that these projects are identified by a folder named Library (see the "Libraries and Robot Projects" topic in Kapow help for details).

When you build the execute request for `RoboServer`, you identify the robot by a robot URL:

```
Request request = new Request("Library:/Input.robot");
```

Here, `Library:/` is a symbolic reference to a robot library, in which the RoboServer should look for the robot. The `RobotLibrary` is specified on the builder:

```
request.setRobotLibrary(new DefaultRobotLibrary());
```

Three different robot library implementations are available, and your selection depends on the deployment environment.

Robot Libraries

Library Type	Description
DefaultRobotLibrary	This library configures RoboServer to look for the robot in the current project folder, which is defined in the Settings application.
	If you have multiple RoboServers, you must deploy your robots on all RoboServers.
	This robot library is not cached, so the robot is reloaded from disk with every execution. This approach makes the library usable in a development environment where robots change often, but it is not suitable for a production environment.

Library Type	Description

EmbeddedFileBasedRobotLibrary	This library is embedded in the execute request sent to RoboServer. To create this library you must create a zip file containing the robots and all dependencies (types, snippets and resources). Use the **Tools** > **Create Robot Library File** menu in Design Studio.
	
	The library is sent with every request, which adds some overhead for large libraries, but the libraries are cached on RoboServer, which offers best possible performance.
	
	One strength is that robots and code can be deployed as a single unit, which offers clean migration from a QA environment to production environment. However, if the robots change often, you will have to redeploy them often.
	
	You can use the following code to configure the embedded robot library for your request.
	
	<pre>Request request = new
	Request("Library:/Tutorials/
	 NewsMagazine.robot");
	RobotLibrary library =
	 new EmbeddedFileBasedRobotLibrary
	 (new FileInputStream
	 ("c:\\embeddedLibrary.robotlib"));
	request.setRobotLibrary(library);</pre>

Library Type	Description
`RepositoryRobotLibrary`	This is the most flexible robot library.
	This library uses the Management Console's built-in repository as a robot library. When you use this library, RoboServer will contact the Management Console, which will send a robot library containing the robot and its dependencies.
	Caching occurs on a per robot basis, inside both Management Console and RoboServer. Inside Management Console, the generated library is cached based on the robot and its dependencies. On RoboServer, the cache is based on a timeout, so it doesn't have to ask the Management Console for each request. In addition, the library loading between RoboServer and Management Console uses HTTP public/ private caching, to further reduce bandwidth.
	If `NewsMagazine.robot` is uploaded to the Management Console, you can use the repository robot library when executing the robot:
| | ```
Request request = new
Request("Library:/Tutorials/
NewsMagazine.robot");
RobotLibrary library = new
RepositoryRobotLibrary("http://
localhost:50080", "Default Project",
60000);
request.setRobotLibrary(library);
``` |
| | This will instruct RoboServer to load the robot from a local Management Console and cache it for one minute before checking with the Management Console to see if a new version of the robot (its type and snippets) has been changed. |
| | In addition, any resource loaded through the `Library:/` protocol causes the RoboServer to request the resource directly from the Management Console. |

Java Advanced

This section describes advanced API features, including output streaming, logging and SSL configuration, as well as parallel execution.

Load Distribution and Failover

Inside the `RequestExecutor`, the executor is given an array of RoboServers. As the executor is constructed, it tries to connect to each RoboServer. Once connected, it sends a ping request to each RoboServer to discover how the server is configured.

Load balanced executor:

```
RoboServer prod = new RoboServer("prod.kapow.local", 50000);
RoboServer prod2 = new RoboServer("prod2.kapow.local", 50000);
Cluster cluster = new Cluster("Prod", new RoboServer[]{ prod, prod2}, false);
```

```
Request.registerCluster(cluster);
```

The load is distributed to each online RoboServer in the cluster, based on the number of unused execution slots on the RoboServer. The next request is always distributed to the RoboServer with the most available slots. The number of available execution slots is obtained through the initial ping response, and the executor keeps track of each robot it starts and when it completes. The number of execution slots on a RoboServer is determined by the **max concurrent robots** setting on the Servers tab.

If a RoboServer goes offline it will not receive any robot execution requests before it has successfully responded to the ping request.

Two Client Rule

You should only have one API client using a given cluster of RoboServer. If you have multiple JVMs running robots against the same RoboServers, it will result in reduced performance.

Executor Logger

When you execute a request, the execute method will throw an exception if a robot generates an error. Other types of errors and warnings are reported through the `ExecutorLogger` interface. In the previous examples, we have not provided any `ExecutionLogger` when executing robots, which means we get the default implementation that will write to System.out. Let's see how the `ExecutorLogger` will report if one of the RoboServers goes offline.

The example configures a cluster with a RoboServer that is not online.

ExecutorLogger, offline server example:

```
RoboServer rs = new RoboServer("localhost", 50000);
Cluster cluster = new Cluster("name", new RoboServer[]{rs}, false);
Request.registerCluster(cluster);
```

If you run this example if should print the following to the console.

ExecutorLogger, offline RoboServer console output:

```
RoboServer{host='localhost', port=50000} went offline.
Connection refused
```

If you do not want to have your application writing directly to System.out, you can provide a different ExecutorLogger implementation, when registering the cluster.

Use DebugExecutorLogger:
```
Request.registerCluster(cluster, new DebugExecutorLogger());
```

This example uses the `DebugExecutorLogger()` which will also print to `System.out`, but only if the API debugging is enabled. You can provide your own implementation of the `ExecutorLogger`, to control how error messages are handled. Check the `ExecutorLogger` JavaDoc for additional details.

Data Streaming

Sometimes you need to present the results from a robot execution in real-time. In these cases, you want the API to return the extracted values immediately instead of waiting for the robot to finish its execution and access the `RQLResult`.

The API offers the possibility to receive a callback every time the API receives a value that was returned by the robot. Do this through the `RobotResponseHandler` interface.

Response streaming using AbstractFailFastRobotResponseHandler:

```
    public class DataStreaming {

    public static void main(String[] args) throws ClusterAlreadyDefinedException {

        RoboServer server = new RoboServer("localhost", 50000);
        Cluster cluster = new Cluster("MyCluster", new RoboServer[] {server}, false);
        Request.registerCluster(cluster);

        try {
            Request request = new Request("Library:/Tutorials/NewsMagazine.robot");
            RobotResponseHandler handler = new AbstractFailFastRobotResponseHandler()
{
                public void handleReturnedValue(RobotOutputObjectResponse response,
Stoppable stoppable) throws RQLException {
                    RQLObject value = response.getOutputObject();
                    Long personId = (Long) value.get("personId");
                    String name = (String) value.get("name");
                    Long age = (Long) value.get("age");
                    System.out.println(personId + ", " + name + ", " + age);
                    }
                };
            request.execute("MyCluster", handler);
        }
    }
 }
```

The preceding example uses the second execute method of the request, which expects a `RobotResponseHandler` in addition to the name of the cluster to execute the robot on. In this example, create a `RobotResponseHandler` by extending `AbstractFailFastRobotResponseHandler`, which provides default error handling, to handle only the values returned by the robot.

The `handleReturnedValue` method is called whenever the API receives a returned value from `RoboServer`. The `AbstractFailFastRobotResponseHandler` used in this example throws exceptions in the same way as the non-streaming execute method. This means that an exception is thrown in response to any API exceptions generated by the robot.

The `RobotResponseHandler` has several methods that can be grouped into three categories.

Robot life cycle events

Methods called when the robot's execution state changes on RoboServer, such as when it starts and finishes execution.

Robot data events

Methods called when the robot returns data or errors to the API.

Additional error handling

Methods called due to an error inside `RoboServer` or in the API.

`RobotResponseHandler` **- robot life cycle events**

Method name	Description
`void requestSent(RoboServer roboServer, ExecuteRequest request)`	Called when the `RequestExecutor` finds the server which will execute the request.
`void requestAccepted(String executionId)`	Called when the found RoboServer accepts the request and puts it into its queue.
`void robotStarted(Stoppable stoppable)`	Called when the RoboServer begins to execute the robot. This usually occurs immediately after the robot is queued, unless the RoboServer is under a heavy load, or used by multiple API clients.
`void robotDone(RobotDoneEvent reason)`	Called when the robot is done executing on RoboServer. The `RobotDoneEvent` is used to specify if the execution terminated normally, due to an error, or if it was stopped.

`RobotResponseHandler` **- robot data events**

Method name	Description
`void handleReturnedValue(RobotOutputObjectResponse response, Stoppable stoppable)`	Called when the robot is executed a Return Value action and the value is returned via the socket to the API.
`void handleRobotError(RobotErrorResponse response, Stoppable stoppable)`	Called when the robot raises an API exception. Under normal circumstances the robot stops executing after the first API exception. This behavior can be overridden by using `Request.setStopRobotOnApiException(false)`, in which case this method is called multiple times. This approach is useful if you want a data streaming robot to continue to execute regardless of any generated errors.
`void handleWriteLog(RobotMessageResponse response, Stoppable stoppable)`	Called when the RoboServer begins to execute the robot. This usually occurs immediately after the robot has been queued, unless the RoboServer is under heavy load, or used by multiple API clients.

`RobotResponseHandler` **- additional error handling**

Method name	Description
`void handleServerError(ServerErrorResponse response, Stoppable stoppable)`	Called if RoboServer generates an error, for instance if the server is too busy to process any requests, or if an error occurs inside RoboServer which prevents it from starting the robot.

Method name	Description
`handleError(RQLException e, Stoppable stoppable)`	Called if an error occurs inside the API, or most commonly, if the client loses the connection to RoboServer.

Many of the methods include a Stoppable object, which can be used to stop in response to a specific error or value returned.

Some methods allow you to throw an `RQLException`, which may have consequences. The thread that calls the handler is the thread that calls `Request.execute()` and exceptions thrown will overload the call stack. If you throw an exception in response to `handleReturnedValue`, `handleRobotError` or `handleWriteLog`, it is your responsibility to invoke `Stoppable.stop()`, or the robot may continue to execute even though the call to `Request.execute()` has completed.

Data streaming is most often used in one of the following use cases.

- Ajax based web applications, where results are presented to the user in real-time. If data is not streamed, results cannot be shown until the robot is done running.
- Robots that return so much data that the client would not be able to hold it all in memory throughout the robot's execution.
- Processes that need to be optimized so the extracted values are processed in parallel with the robot execution.
- Processes that store data in databases in a custom format.
- Robots that should ignore or require custom handling of API exceptions (see the following).

Response and error collecting using AbstractFailFastRobotResponseHandler:

```
public class DataStreamingCollectErrorsAndValues {

 public static void main(String[] args) throws ClusterAlreadyDefinedException {

  RoboServer server = new RoboServer("localhost", 50000);
  Cluster cluster = new Cluster("MyCluster", new RoboServer[] {server}, false);
  Request.registerCluster(cluster);

  try {
    Request request = new Request("Library:/Tutorials/NewsMagazine.robot");
    request.setStopRobotOnApiException(false);  // IMPORTANT!!
    request.setRobotLibrary(new DefaultRobotLibrary());
    ErrorCollectingRobotResponseHandler handler =
      new ErrorCollectingRobotResponseHandler();
    request.execute("MyCluster", handler);

    System.out.println("Extracted values:");
    for (RobotOutputObjectResponse response : handler.getOutput()) {
        RQLObject value = response.getOutputObject();
        Long personId = (Long) value.get("personId");
        String name = (String) value.get("name");
        Long age = (Long) value.get("age");
        System.out.println(personId + ", " + name + ", " + age);
     }

    System.out.println("Errors:");
    for (RobotErrorResponse error : handler.getErrors()) {
        System.out.println(error.getErrorLocationCode() + ", " +
          error.getErrorMessage());
```

```
        }
    }
}

private static class ErrorCollectingRobotResponseHandler extends
        AbstractFailFastRobotResponseHandler {

    private List<RobotErrorResponse> _errors =
        new LinkedList<RobotErrorResponse>();
    private List<RobotOutputObjectResponse> _output =
        new LinkedList<RobotOutputObjectResponse>();
    public void handleReturnedValue
        (RobotOutputObjectResponse response, Stoppable stoppable)
        throws RQLException {
      _output.add(response);
}

@Override
public void handleRobotError(RobotErrorResponse response,
      Stoppable stoppable) throws RQLException {
    // do not call super as this will stop the robot
    _errors.add(response);
}

public List<RobotErrorResponse> getErrors() {
      return _errors;
}

public List<RobotOutputObjectResponse> getOutput() {
      return _output;
    }
  }
}
```

The preceding example shows how to use a `RobotResponseHandler` that collects returned values and errors. This type of handler is useful if the robot should continue to execute even when errors are encountered, which can be useful if the website is unstable and occasionally times out. Notice that only robot errors (API exceptions) are collected by the handler. If the connection to `RoboServer` is lost `Request.execute()` will still throw an `RQLException` (and the robot will be stopped by `RoboServer`).

For more details, check the `RobotResponseHandler` JavaDoc.

SSL

The API communicates with RoboServer through an `RQLService`, which is a RoboServer component that listens for API requests on a specific network port. When you start a RoboServer, you specify if it should use the encrypted SSL service, or the plain socket service, or both (using two different ports). All RoboServers in a cluster must be running the same `RQLService` (although the port may be different).

Assuming we have started a RoboServer with the SSL `RQLService` on port 50043:

```
RoboServer -service ssl:50043
```

we can use the following code.

SSL configuration

```
RoboServer server = new RoboServer("localhost", 50043);
boolean ssl = true;
```

```
Cluster cluster = new Cluster("MyCluster", new RoboServer[] {server}, ssl);
Request.registerCluster(cluster);
```

All we need to do is to create the cluster as an SSL cluster and specify the SSL port used by each RoboServer. Now all communication between `RoboServer` and the API will be encrypted.

For this example to work, you need `not-yet-commons-ssl-0.3.17.jar` in you application `classpath`. You can find it next to the API jar file inside your Kapow installation.

In addition to data encryption, SSL offers the possibility to verify the identity of the remote party. This type of verification is very important on the Internet, as rogue Web sites could otherwise pretend to be someone they are not. Most often your API client and RoboServers are on the same local network, so you rarely need to verify the identity of the other party, but the API supports this feature should it become necessary.

Because identity verification is almost never used, it is not described in this guide. If you are interested, see the SSL examples included with the Java API.

Parallel Execution

Both execute methods of the Request are blocking, which means that a thread is required for each robot execution. The examples we have looked at until now have all executed the robot directly on the main thread, which is typically not preferable as you can only execute a single robot at a time in a sequential manner.

The following example executes two tutorial robots in parallel. This example uses the `java.util.concurrent` library for multithreading.

Multithreading Example

```
import com.kapowtech.robosuite.api.java.repository.construct.*;
import com.kapowtech.robosuite.api.java.rql.*;
import com.kapowtech.robosuite.api.java.rql.construct.*;
import com.kapowtech.robosuite.api.java.rql.engine.hotstandby.*;

import java.util.concurrent.*;

public class ParallelExecution {

    public static void main(String[] args) throws Exception {

        RoboServer server = new RoboServer("localhost", 50000);
        Cluster cluster = new Cluster("MyCluster", new RoboServer[] {server},
         false);
        Request.registerCluster(cluster);

        int numRobots = 4;
        int numThreads = 2;
        ThreadPoolExecutor threadPool = new ThreadPoolExecutor(numThreads,
          numThreads, 10, TimeUnit.SECONDS, new LinkedBlockingQueue());
        for (int i = 0; i < numRobots; i++) {
          Request request = new Request("Library:/Tutorials/NewsMagazine.robot");
          request.setRobotLibrary(new DefaultRobotLibrary());
          threadPool.execute(new RobotRunnable(request));
        }
        threadPool.shutdown();
        threadPool.awaitTermination(60, TimeUnit.SECONDS);
    }
```

```
    // ---------------------------------------------------------------
    // Inner classes
    // ---------------------------------------------------------------
    static class RobotRunnable implements Runnable {

        Request _request;

        RobotRunnable(Request request) {
            _request = request;
        }

        public void run() {

            try {
                RQLResult result = _request.execute("MyCluster");
                System.out.println(result);
            }
        }
    }
}
```

The preceding example creates a `ThreadPoolExecutor` with two threads, and we then create four `RobotRunnables` and execute them on the thread pool. Since the thread pool has two threads, two robots start to execute immediately. The remaining two robots are parked in the `LinkedBlockingQueue` and executed in order after the two first robot finish their execution and the thread pool threads become available.

Please note that the request is mutable, and to avoid raising conditions, the request is cloned inside the execute method. Because a request is mutable, you should never modify the same request on separate threads.

Repository Integration

In the Management Console you also specify clusters of RoboServers, which are used to execute scheduled robots, as well as robots executed as REST services. The API allows you to use the `RepositoryClient` to obtain cluster information from Management Console. See the `RepositoryClient` documentation for details.

Repository Integration:

```
    public class RepositoryIntegration {
    public static void main(String[] args) throws Exception {

     RepositoryClient client = RepositoryClientFactory.createRepositoryClient
       ("http://localhost:50080", null, null);
     Request.registerCluster(client, "Cluster 1");

     Request request = new Request("Library:/Tutorials/NewsMagazine.robot");
     request.setRobotLibrary(new DefaultRobotLibrary());
     RQLResult result = request.execute("MyCluster");
     System.out.println(result);
    }
}
```

The preceding example shows how to create a `RepositoryClient` that connects to a Management Console deployed on localhost. For this example to work, you must have `commons-`

logging-1.1.1.jar, commons-codec-1.4.jar, and commons-httpclient-4.1.jar included in your classpath.

Authentication is not enabled so null is passed for both username and password. When we register the RepositoryClient we specify the name of a cluster that exists on the Management Console. This will then query the Management Console to get a list of RoboServers configured for this cluster, and check every 2 minutes to see if the cluster configuration has been updated on the Management Console.

This integration allows you to create a cluster on Management Console that you can change dynamically using the Management Console user interface. When you use a Management Console cluster with the API usage should be exclusive, and you should not use it for scheduling robot, as this would break the two client rule.

Under the Hood

The section will explain what is going on under the hood when you register a cluster and execute Requests.

When you register a cluster with the request, a RequestExecutor is created behind the scene. This RequestExecutor is stored in a Map using the cluster name as key. When a request is executed, the provided cluster name is used to find the associated RequestExecutor and execute the request.

Let's look at a short example.

Normal execution

```
public static void main(String[] args) throws InterruptedException,
    RQLException {

RoboServer server = new RoboServer("localhost", 50000);
Cluster cluster = new Cluster("MyCluster", new RoboServer[]{ server}, false);
Request.registerCluster(cluster);
Request request = new Request("Library:/Tutorials/NewsMagazine.robot");
request.setRobotLibrary(new DefaultRobotLibrary());
RQLResult result = request.execute("MyCluster");
System.out.println(result);
}
```

Now write the same example by using the hiddenRequestExecutor directly.

Under the hood execution:

```
public static void main(String[] args) throws InterruptedException,
    RQLException {

 RoboServer server = new RoboServer("localhost", 50000);
 Cluster cluster = new Cluster("MyCluster", new RoboServer[]{ server}, false);
 RequestExecutor executor = new RequestExecutor(cluster);

 Request request = new Request("Library:/Tutorials/NewsMagazine.robot");
 request.setRobotLibrary(new DefaultRobotLibrary());
 RQLResult result = executor.execute(request);
 System.out.println(result);
 }
```

The `RequestExecutor` is hidden by default, so you don't have to keep track of it. You may only create one `RequestExecutor` per cluster, so if you use it directly you need to store a reference to it throughout your application. Using `Request.registerCluster(cluster)` means that you can ignore the `RequestExecutor` and lifecycle rules.

The `RequestExecutor` contains the necessary state and logic which provides the load balancing and failover features. Using the `RequestExecutor` directly also offers a few extra features.

RequestExecutor Features

When the `RequestExecutor` is not connected to a repository, you can dynamically add or remove RoboServers, by calling `addRoboServer(..)` and `removeRoboServer(..)`. These methods modify the distribution list used inside the `RequestExecutor`.

`RequestExecutor.getTotalAvailableSlots()` returns the number of unused execution slots across all `RoboServers` in the internal distribution list.

By using these methods, you can dynamically add `RoboServers` to your `RequestExecutor` once the number of available execution slots becomes low.

When you create the `RequestExecutor`, you may optionally provide an `RQLEngineFactory`. The `RQLEngineFactory` allows you to customize which `RQLProtocol` is used when connecting to a `RoboServer`. This is only needed under very rare circumstances. For example, if you want to use a client certificate to increase security (see the Certificates chapter in the *Kapow Administrator's Guide* for details).

Web Applications

The `RequestExecutor` contains a number of internal threads used for sending and receiving requests to RoboServers, as well as pinging each known RoboServer at regular intervals. These threads are all marked as daemon, which means that they don't prevent the JVM from stopping when the main thread exists. See Thread JavaDoc for details on daemon threads.

If you use the `RequestExecutor` inside a web application, the JVM has a longer lifespan than your web application, and you can deploy and undeploy your web application while the web container is running. This means that a web application is responsible for stopping any threads that it has created. If the web application does not stop a thread, a memory leak is created when you undeploy the web application. The memory leak occurs because any objects referenced by running threads cannot be garbage collected until the threads stop.

If you use the `RequestExecutor` inside a web application, your code is responsible for shutting down these internal threads, this is done by calling `Request.shutdown()` or `RequestExecutor.shutdown()` if your code created the `RequestExecutor` explicitly.

This example shows you how to use a `ServletContextListener` to shut down the API correctly when a web application is undeployed. You must define the context listener in your application's web.xml.

Proper shutdown in web application:

```
import com.kapowtech.robosuite.api.java.repository.construct.*;
import com.kapowtech.robosuite.api.java.rql.*;
import com.kapowtech.robosuite.api.java.rql.construct.*;
```

```
import javax.servlet.*;

public class APIShutdownListener implements ServletContextListener {
    public void contextInitialized(ServletContextEvent servletContextEvent) {
        RoboServer server = new RoboServer("localhost", 50000);
        Cluster cluster = new Cluster("MyCluster", new RoboServer[]{ server},
          false);
        try {
            Request.registerCluster(cluster);
        }
        catch (ClusterAlreadyDefinedException e) {
            throw new RuntimeException(e);
        }
    }

    public void contextDestroyed(ServletContextEvent servletContextEvent) {
        Request.shutdown();
    }
}
```

`contextDestroyed` is called when the web container un-deploys the application. Here we call Request.shutdown() to ensure that all internal threads in the hidden `RequestExecutor` are stopped correctly.

Since contextInitialized can't throw any unchecked exceptions we have to wrap the `ClusterAlreadyDefinedException` in a `RunTimeException`. Developers may be tempted to ignore the `ClusterAlreadyDefinedException` at this location, because they claim that it cannot be thrown, as our application has not defined any other clusters. However, due to the class loader hierarchy in Java web containers it is actually possible to get this exception if the application is deployed twice. It will only occur if the API jar file was loaded by a common class loader and not by the individual application's class loader.

API Debugging

Although this is rarely needed, the API can provide additional information for debugging purposes. To enable API debugging, you need to configure the system property `DEBUG_ON`. The value of this property must be a package/class name in the API.

For instance, if you are interested in the data transmissions between the API and `RoboServer`, you could ask for debugging information for package `com.kapowtech.robosuite.api.java.rql.io`. While you are developing, do this by directly setting the system property in code:

Enabling Debug:

```
System.setProperty("DEBUG_ON", "com.kapowtech.robosuite.api.java.rql.io");
RoboServer server = new RoboServer("localhost", 50000);
Cluster cluster = new Cluster("MyCluster", new RoboServer[]{ server}, false);
Request.registerCluster(cluster);
```

If you are debugging an application in production, you would define the system property via the command line.

Enabling Debug:

```
java -DDEBUG_ON=com.kapowtech.robosuite.api.java.rql.io Tutorial1
```

If you are interested in debugging from multiple packages, separate the package names by commas. Instead of a package name, you can provide the argument ALL, to have debugging from all packages printed.

Repository API

The Repository API allows you to query the Management Console's Repository, to get a list of projects, robots and the input required to call a robot. It also allows you to programmatically deploy robots, types and resource files.

Dependencies

To use the Repository API, you need the following libraries in the `API/robosuite-java-api/lib` folder inside your Kofax Kapow installation folder.

- `commons-codec-1.4.jar`
- `commons-httpclient-4.1.jar`, or newer
- `commons-logging-1.1.1.jar`
- `not-yet-commons-ssl-0.3.17.jar` or newer: if your Management Console must be accessed through HTTPS
- `dom4j-1.6.1.jar`
- `xstream-1.4.7.jar`

Use Java 8 or later.

Repository Client

Communication with the repository is achieved through the `RepositoryClient` in the `com.kapowtech.robosuite.api.java.repository.engine`.

Create RepositoryClient:

```
    public static void main(String[] args) {

    String username = "admin";
    String password = "admin";
    try {
        RepositoryClient client = RepositoryClientFactory.
            createRepositoryClient("http://localhost:50080/",
            username, password);
        Project[] projects = client.getProjects();
        for (Project project : projects) {
            System.out.println(project.getName());
        }
    }
 }
```

Here we see a `RepositoryClient` configured to connect to Management Console's repository on `http://localhost:50080/`, with a username and password.

Once the `RepositoryClient` is created, we use the `getProjects()` method to query the repository for a list of projects. Notice that when calling any of the `RepositoryClient` methods, a `RepositoryClientException` is thrown if an error occurs.

The `RepositoryClient` has the following eleven methods.

Methods of the RepositoryClient:

Method signature	Description
`void deleteResource(String projectName, String resourceName, boolean silent)`	Deletes a resource from a project. If silent is true, no error is generated if the resource does not exist. The `resourceName` argument uses the full path of the resource.
`void deleteRobot(String projectName, String robotName, boolean silent)`	Deletes a robot from a project. The `robotName` argument uses the full path of the robot.
`void deleteSnippet(String projectName, String snippetName, boolean silent)`	Deletes a snippet from a project. The `snippetName` argument uses the full path of the snippet.
`void deleteType(String projectName, String modelName, boolean silent)`	Deletes a type from a project. The `modelName` argument uses the full path of the type.
`void deployLibrary(String projectName, EmbeddedFileBasedRobotLibrary library, boolean failIfExists)`	Deploys a library to the server. Robots, types and resources are overridden unless `failIfExists` is true.
`void deployResource(String projectName, String resourceName, byte[] resourceBytes, boolean failIfExists)`	Deploys a resource to a project. If a resource with the given name already exists, it can be overridden by setting `failIfExists` to false. The `resourceName` argument uses the full path of the resource.
`void deployRobot(String projectName, String robotName, byte[] robotBytes, boolean failIfExists)`	Deploys a robot to a project. If a robot with the given name already exists, it can be overridden by setting `failIfExists` to false. The `robotName` argument uses the full path of the robot.
`void deploySnippet(String projectName, String snippetName, byte[] snippetBytes, boolean failIfExists)`	Deploys a snippet to a project. If a snippet with the given name already exists, it can be overridden by setting `failIfExists` to false. The `snippetName` argument uses the full path of the snippet.
`void deployType(String projectName, String typeName, byte[] typeBytes, boolean failIfExists)`	Deploys a type to a project. If a type with the given name already exists, it can be overridden by setting `failIfExists` to false. The `typeName` argument uses the full path of the type.
`Project[] getProjects()`	Returns the projects that exist in this repository.
`Cluster[] getRoboServerClusters()`	Returns a list of clusters and online(valid) RoboServers that are registered with the Management Console running the repository.
`Cluster[] getRoboServerClusters(boolean onlineRoboServer)`	Returns a list of clusters and RoboServers that are registered with the Management Console. Use the `onlineRoboServer` flag to indicate if the list of clusters should include only RoboSevers that are online or all of the RoboServers.

Method signature	Description
`Cluster addRoboServer(String clusterName, int portNumber, String host)`	Adds a new RoboServer to a cluster.
`Robot[] getRobotsInProject(String projectName)`	Returns the full path of the robots available in the project.
`RobotSignature getRobotSignature(String projectName, String robotName)`	Returns the robot signature with the full path of the robot, as well as the input variables required to execute this robot and a list of the types it may return or store.
`RepositoryFolder getProjectInventory(String projectName)`	Returns the entire tree of folders and files from the repository.
`RepositoryFolder getFolderInventory(String projectName, String folderPath)`	Returns the folders and files of the subfolder in the specified project from the repository.
`RepositoryFolder getFileInventory(String projectName, String folderPath, String fileName, RepositoryFile.Type fileType)`	Gets the file and the referenced files from the management console. Note that the file inventory is wrapped in a RepositoryFolder to get references.
`void deleteFile(RepositoryFile file, bool silent)`	Deletes the specified file from the repository.
`Date getCurrentDate()`	Returns current date and time of the Management Console.
`byte[] getBytes(RepositoryFile file)`	Returns the size in bytes of the specified file in the repository.
`String computeChecksum(byte[] bytes)`	Returns the checksum of the specified file to verify data integrity.
`void updateFile(RepositoryFile file, byte[] bytes)`	Updates the specified file in the repository with new bytes.
`void moveFile (RepositoryFile sourceFile, String destFolderPath)`	Moves the specified file from the repository to a folder specified in `destFolderPath`.
`void renameRobot(RepositoryFile robotFile, String newName)`	Renames the specified robot file.
`void deleteFolder(String projectName, String folderPath)`	Deletes the specified folder in the repository.
`void deleteRoboServer(String clusterName, RoboServer roboServer)`	Deletes a RoboServer.

Method signature	Description
`Map<String, String> getInfo()`	Returns information about the Management Console and the Repository API The method returns a mapping of the following: • "application" to the version of the Management Console containing major, minor and dot version, for example, `10.3.2` • "repository" to the ID of the latest DTD used by the Repository API, such as: `//Kapow Technologies//DTD Repository 1.5//EN` • "rql" to the ID of the latest DTD used by the Robot Query Language API, such as: `//Kapow Technologies//DTD RoboSuite Robot Query Language 1.13//EN`

Note The full path is relative to your project folder.

Proxy servers must be specified explicitly when creating the `RepositoryClient`. Standard http proxy servers without authentication are supported. NTLM proxy servers with authentication are also supported.

Check the `RepositoryClient` JavaDoc for additional details.

Deployment via Repository Client

The following example shows how to deploy a robot and a type from the local file system using the `RepositoryClient`.

Deployment using RepositoryClient:

```
String user = "test";
 String password = "test1234";
 RepositoryClient client = new RepositoryClient("http://localhost:50080", user,
     password);
 try {
     FileInputStream robotStream = new FileInputStream
      ("c:\\MyRobots\\Library\\Test.robot");
     FileInputStream typeStream = new FileInputStream
      ("c:\\MyRobots\\Library\\Test.type");

     // Use the Kapow Java APIs StreamUtil to convert InputStream to byte[].
     // For production we recommend IOUtils.toByteArray(InputStream i)
        in the commons-io library from apache.
     byte[] robotBytes = StreamUtil.readStream(robotStream).toByteArray();
     byte[] typeBytes = StreamUtil.readStream(typeStream).toByteArray();

     // we assume that no one has deleted the Default project
     client.deployRobot("Default project", "Test.robot", robotBytes, true);
     client.deployType("Default project", "Test.type", typeBytes, true);
 }
 catch (FileNotFoundException e) {
     System.out.println("Could not load file from disk " + e.getMessage());
 }
 catch (IOException e) {
     System.out.println("Could not read bytes from stream " + e.getMessage());
```

```
}
catch (FileAlreadyExistsException e) {
    // either the type or file already exist in the give project
    System.out.println(e.getMessage());
}
```

Repository Rest API

The repository API is actually a group of restful services (and URLs where data can be posted).

All the repository client methods that retrieve information from the repository send XML to the Repository, and the Repository responds with XML. All deploy methods post bytes to the Repository (information encoded in URL) and the Repository returns XML to acknowledge. The format of the XML sent and received is governed by a DTD found at www.kapowtech.com.

Here is an example of all the XML based requests. All messages must start with the following declaration:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE repository-request PUBLIC "-//Kapow Technologies//
DTD Repository 1.3//EN" "http://www.kapowtech.com/robosuite/
repository_1_5.dtd">
```

If the Management Console is deployed at `http://localhost:8080/ManagementConsole`, the requests must be posted to `http://localhost:8080/ManagementConsole/secure/RepositoryAPI?format=xml`

Snippets

A number of XML snippets are used throughout the API and the following are snippets used in the examples. We recommend studying the DTD to understand the structure of the data.

When sending requests, we often need to describe a file. Similarly, responses contain data about a file. The following table shows snippets that are found shortened in the examples. The constructs have been added to the 1.5 DTD to assist in project synchronization between Design Studio and Management Console.

Snippet Name	Code
repository-file-request	`<repository-file-request> <project-name>Default project</project-name> <name>ExName</name> <type>snippet</type> <path>subfolder</path> <last-modified>2015-02-01 19:26:12.321</last-modified> <last-modified-by>username</last-modified-by> <checksum>a342ddaf</checksum> </repository-file-request>`
repository-file	`<repository-file><name>filename</name> <type>ROBOT</name><last-modified>2015-02-01 19:26:12.321</last-modified><last-modified-by>username</last-modified-by><checksum>a342ddaf</checksum><dependencies><dependency><name>exsnippet</name><type>snippet</type></dependency> </dependencies></repository-file>`

REST Operations

Method	Example Request	Example Response
delete-file (robot)	`<repository-request> <delete-file file-type="robot" silent="true"> <project-name>Default project</project-name> <file-name>InputA.type</file-name> </delete-file> </repository-request>`	`<repository-response><delete-successful/></repository-response>`
delete-file (type)	`<repository-request> <delete-file file-type="type" silent="false"> <project-name>Default project</project-name> <file-name>InputA.type</file-name> </delete-file> </repository-request>`	`<repository-response><error type="file-not-found">Could not find a Type named InputA.type in project 'Default project'</error></repository-response>`
delete-file (snippet)	`<repository-request> <delete-file file-type="snippet" silent="true"> <project-name>Default project</project-name> <file-name>InputA.type</file-name> </delete-file> </repository-request>`	`<repository-response><delete-successful/></repository-response>`
delete-file (resource)	`<repository-request> <delete-file file-type="resource" silent="true"> <project-name>Default project</project-name> <file-name>InputA.type</file-name> </delete-file> </repository-request>`	`<repository-response><delete-successful/></repository-response>`
get-projects	`<repository-request> <get-projects/> </repository-request>`	`<repository-response><project-list><project-name>Default project</project-name></project-list></repository-response>`
get-robots-in-project	`<repository-request> <get-robots-in-project> <project-name>Default project</project-name> </get-robots-in-project> </repository-request>`	`<repository-response><robot-list><robot><robot-name>DoNothing.robot</robot-name><version>7.2</version><last-modified>2011-10-11 18:24:12.648</last-modified></robot></robot-list></repository-response>`

Method	Example Request	Example Response
get-robot-signature	`<repository-request> <get-robot-signature> <project-name>Default project</project-name> <robot-name>DoNothing.robot</robot-name> </get-robot-signature> </repository-request>`	`<repository-response><robot-signature><robot-name>DoNothing.robot</robot-name><version>7.2</version><last-modified>2011-10-11 18:24:12.648</last-modified><input-object-list><input-object><variable-name>InputA</variable-name><type-name>InputA</type-name><input-attribute-list><input-attribute><attribute-name>aString</attribute-name><attribute-type>Short Text</attribute-type></input-attribute><input-attribute><attribute-name>anInt</attribute-name><attribute-type>Integer</attribute-type></input-attribute><input-attribute><attribute-name>aNumber</attribute-name><attribute-type>Number</attribute-type></input-attribute><input-attribute><attribute-name>aSession</attribute-name><attribute-type>Session</attribute-type></input-attribute><input-attribute><attribute-name>aBoolean</attribute-name><attribute-type>Boolean</attribute-type></input-attribute><input-attribute><attribute-name>aDate</attribute-name><attribute-type>Date</attribute-type></input-attribute><input-attribute><attribute-name>aCharacter</attribute-name><attribute-type>Character</attribute-type></input-attribute><input-attribute><attribute-name>anImage</attribute-name><attribute-type>Image</attribute-type></input-attribute></input-attribute-list></input-object><input-object><variable-name>InputB</variable-name><type-name>InputB</type-name><input-attribute-list><input-attribute required="true"><attribute-name>aString</attribute-`

Method	Example Request	Example Response
`get-clusters`	`<repository-request> <get-clusters/> </repository-request>`	`<repository-response><clusters><cluster name="Cluster 1" ssl="false"><roboserver host="localhost" port="50000"/></cluster></clusters></repository-response>`
`get-current-date`	`<repository-request> <get-current-date/> </repository-request>`	`<repository-response> <current-date>2015-02-01 19:26:12.321</current-date> </repository-response>`
`get-bytes`	`<repository-request> <get-bytes> <repository-file-request>EXAMPLE</repository-file-request> </get-bytes> </repository-request>`	`<repository-response> <file-content> <file-bytes><![CDATA[ --- BASE 64 Encoded file content --- ]]> </file-bytes> </file-content> </repository-response>`
`get-project-inventory`	`<repository-request> <get-project-inventory> <project-name>Default project</project-name> </get-project-inventory> </repository-request>`	`<repository-response> <repository-folder> <path></path> <sub-folders> -- repository-folders (recursively) -- </sub-folders> <files> -- zero, one or more repository-file elements -- </files> <references> -- zero, one or more repository-file elements needed by robots in folder -- </references> </repository-folder> </repository-response>`
`get-folder-inventory`	`<repository-request> <get-folder-inventory> <project-name>Default project</project-name> <path>subfolder</path> </get-folder-inventory> </repository-request>`	`<repository-response> <repository-folder> <path></path> <sub-folders> -- repository-folders (recursively) -- </sub-folders> <files> -- zero, one or more repository-file elements -- </files> <references> -- zero, one or more repository-file elements needed by robots in folder -- </references> </repository-folder> </repository-response>`

Method	Example Request	Example Response
get-file-inventory	`<repository-request> <get-file-inventory> <project-name>Default project</project-name> <path>subfolder</path> <name>robotname</name> <type>robot</type> </get-file-inventory> </repository-request>`	`<repository-response> <repository-folder> <path></path> <sub-folders> -- repository-folders (recursively) -- </sub-folders> <files> -- zero, one or more repository-file elements -- </files> <references> -- zero, one or more repository-file elements needed by robots in folder -- </references> </repository-folder> </repository-response>`
update-file	`<repository-request> <update-file> <repository-file-request>...</repository-file-request> <file-bytes><![CDATA[ --- BASE 64 Encoded file content ---]]> </update-file> </repository-request>`	`<repository-response> <update-successful/> </repository-response>`
get-clusters	`<repository-request> <get-clusters online-roboserver='true'/> </repository-request>`	`<repository-response> <clusters> <cluster name='ClusterName' ssl='false'> <roboserver host='localhost' port='50000' primary='true'/> </cluster> </clusters> </repository-response>`
add-roboserver	`<repository-request> <add-roboserver> <cluster name='ClusterName' ssl='false'> <roboserver host='localhost' port='50000' primary='true'/> </cluster> <roboserver host='localhost' port='50001' primary='true'/> </add-roboserver> </repository-request>`	`<repository-response> <clusters> <cluster name='ClusterName' ssl='false'> <roboserver host='localhost' port='50000' primary='true'/> <roboserver host='localhost' port='50001' primary='true'/> </cluster> </clusters> </repository-response>`
delete-roboserver	`<repository-request> <add-roboserver> <cluster name='ClusterName' ssl='false'> <roboserver host='localhost' port='50000' primary='true'/> <roboserver host='localhost' port='50001' primary='true'/> </cluster> <roboserver host='localhost' port='50001' primary='true'/> </add-roboserver> </repository-request>`	`<repository-response> <cluster name='ClusterName' ssl='false'> <roboserver host='localhost' port='50000' primary='true'/> </cluster> </repository-response>`

Method	Example Request	Example Response
delete-folder	`<repository-request> <delete-folder> <project-name>Default project</project-name> <path>path/to/empty/folder</path> </delete-folder> </repository-request>`	`<repository-response> <delete-successful/> </repository-response>`
move-file	`<repository-request> <move-file> <repository-file-request>...</repository-file-request> <path>new/destination/path</path> </move-file> </repository-request>`	`<repository-response> <update-successful/> </repository-response>`
Rename-robot	`<repository-request> <rename-robot> <repository-file-request>...</repository-file-request> <file-name>newnameofrobot</file-name> </rename-robot> </repository-request>`	`<repository-response> <update-successful/> </repository-response>`

Note Robot, Type, Snippet, and Resource names must be specified as full path. The full path is relative to your project folder.

The deployment is done by posting the raw bytes (the octet-stream is sent as a post body) to the following URLs. Here is an example where the repository is deployed on http://localhost:8080/ManagementConsole

Methods of the deploy operations:

Operation	URL
deploy robot	`http://localhost:8080/ManagementConsole/secure/RepositoryAPI?format=bytes&operation=deployRobot&projectName=Default project&fileName=DoNothing.robot&failIfExists=true`
deploy type	`http://localhost:8080/ManagementConsole/secure/RepositoryAPI?format=bytes&operation=deployType&projectName=Default project&fileName=InputA.type&failIfExists=true`
deploy Snippet	`http://localhost:8080/ManagementConsole/secure/RepositoryAPI?format=bytes&operation=deploySnippet&projectName=Default project&fileName=A.snippet&failIfExists=true`
deploy resource	`http://localhost:8080/ManagementConsole/secure/RepositoryAPI?format=bytes&operation=deployResource&projectName=Default project&fileName=resource.txt&failIfExists=true`
deploy library	`http://localhost:8080/ManagementConsole/secure/RepositoryAPI?format=bytes&operation=deployLibrary&projectName=Default project&fileName=NA&failIfExists=true`

If authentication is enabled on Management Console, the URL `http://localhost:8080/ManagementConsole/secure/RepositoryAPI` is protected by basic authentication.

This allows you to include credentials in the URL in the following manner: `http://username:password@localhost:8080/ManagementConsole/secure/RepositoryAPI`.

Chapter 2

.NET Programmer's Guide

This chapter describes how to execute Robots using the Kapow .NET API. The guide assumes that you have completed the Design Studio tutorials and know how to write simple Robots, and that you are familiar with the C# programming language.

Details about specific classes are found in the compiled help, `robosuite-dotnet-api.chm` located in `\API\robosuite-dotnet-api\docs` inside the Kapow installation folder.

.Net Basics

By using the .NET API, any .NET-based application (.NET 4.0 required) can become a client to a RoboServer. In addition to running robots that store data in a database, you can also have the robots return data directly back to the client application. Here are some examples:

- Use multiple robots to do a search that aggregates results from multiple sources in real time.
- Run a robot in response to an event on your application back end. For instance, run a robot when a new user signs up to create accounts on web-based systems not integrated directly into your back end.

This guide introduces the core classes, and how to use them for executing robots. It also describes how to provide input to robots and control their execution on a RoboServer.

The .NET API is a .dll file that is located in `/API/robosuite-dotnet-api/lib/robosuite-dotnet-api.dll` inside the Kofax Kapow installation folder (see the "Important Folders in Kapow" topic in the *Installation Guide* for details). All examples in this guide can be found in `/API/robosuite-dotnet-api/examples`. `log4net.dll` is a required third-party library located next to the .NET API file.

First Example

Let's start by looking at the code required to execute the robot named `NewsMagazine.robot`, which is located in the `Tutorials` folder of the default project. The robot outputs its results using the Return Value step action, which makes it easy to handle the output programmatically using the API. Other robots (typically those run in a schedule by the Management Console) store their data directly in a database using the Store in Database step action, in which case data collected by the robot will not be returned to the API client.

In the following example, we will look at how to execute the `NewsMagazine` robot and process the output programmatically.

Execute a Robot without input:

```
using System;
using System.Collections.Generic;
using System.Text;
```

```
using Com.KapowTech.RoboSuite.Api;
using Com.KapowTech.RoboSuite.Api.Repository.Construct;
using Com.KapowTech.RoboSuite.Api.Construct;

namespace Examples
{
  class Program
  {
      static void Main(string[] args)
      {
          var server = new RoboServer("localhost", 50000);
          var ssl = false;
          var cluster = new Cluster("MyCluster", new RoboServer[]{ server}, ssl);

          Request.RegisterCluster(cluster); // you can only register a cluster
              once per application

          var request = new Request("Library:/Tutorials/NewsMagazine.robot");
          request.RobotLibrary = new DefaultRobotLibrary();
          RqlResult result = request.Execute("MyCluster");

          foreach (RqlObject value in result.GetOutputObjectsByName("Post")) {
              var title = value["title"];
              var preview = value["preview"];
              Console.WriteLine(title + ", " + preview);
          }
          Console.ReadKey();
      }
  }
}
```

Let's start by looking at the classes involved and their responsibilities.

RoboServer	This is a simple value object that identifies a RoboServer which can execute robots. Each RoboServer must be activated by a Management Console and assigned KCU before use.
Cluster	A cluster is a group of RoboServer functioning as a single logical unit.
Request	This class is used to construct the robot request. Before you can execute any requests you must register a cluster with the Request class.
DefaultRobotLibrary	A robot library instructs RoboServer where to find the robot identified in the request. Later examples will explore the various robot library types and when/how to use them.
RQLResult	This contains the result of a robot execution. The result contains value responses, log and server messages.
RQLObject	Each value that is returned from a robot using the Return Value action can be accessed as an RQLObject.

Now let's go through each line in the example and look at the specifics.

The first line tells the API that our RoboServer is running on localhost port 50000.

```
var server = new RoboServer("localhost", 50000);
```

The following lines define a cluster with a single RoboServer. The cluster is registered with the Request class, allowing you to execute request on this cluster. Each cluster can only be registered once per application, which is done during the initialization of the application.

Registering a cluster:

```
var ssl = false;
 var cluster = new Cluster("MyCluster", new RoboServer[]{ server}, ssl);
 Request.RegisterCluster(cluster);
```

The followed code creates a request that executes the robot named `NewsMagazine.robot` located at `Library:/Tutorials Library:/` refers to the robot Library configured for the request. Here the `DefaultRobotLibrary` is used, which instructs the RoboServer to look for the robot in the servers local file system. See Robot Libraries for details on how to use robot libraries.

```
var request = new Request("Library:/Tutorials/NewsMagazine.robot");
 request.RobotLibrary = new DefaultRobotLibrary();
```

The next line executes the robot on the cluster named `MyCluster` (the cluster we previously registered) and returns the result once the robot is done. If an error occurs while the robot is executing an exception will be thrown here.

```
RqlResult result = request.Execute("MyCluster");
```

Finally we process the extracted values. First we get all extracted values of the type named Post , and iterate through them. For each `RQLObject` we access the attributes of the Post type, and print the result. We will look at attributes and mappings in a later section.

```
    foreach (RqlObject value in result.GetOutputObjectsByName("Post")) {
    var title = value["title"];
    var preview = value["preview"];
    Console.WriteLine(title + ", " + preview);
```

Robot Input

Most robots executed through the API will be parametrized through input, such as a search keyword, or login credentials. Input to a robot is part of the request to RoboServer, and is provided using the `createInputVariable` method on the request. Let us look at a short code fragment.

Input using implicit RQLObjectBuilder

```
var request = new Request("Library:/Tutorials/Input.robot");
request.CreateInputVariable("userLogin").SetAttributeEntry
  ("username", "scott").SetAttributeEntry("password", "tiger");
```

In the preceding code we create a `Request` and use `CreateInputVariable` to create an input variable named `userLogin`. We then use `setAttribute` to configure the username and password attributes of the input variable.

The preceding example is a common shorthand notation, but can also be expressed in move detail by using the `RqlObjectBuilder`:

```
var request = new Request("Library:/NewsMagazine.robot");
RqlObjectBuilder userLogin = request.CreateInputVariable("userLogin");
userLogin.SetAttributeEntry("username", "scott");
userLogin.SetAttributeEntry("password", "tiger");
```

The two examples are identical. The first utilizes the cascading method invocation on the anonymous `RqlObjectBuilder` and is therefore shorter.

When RoboServer receives this request the following occurs:

- RoboServer loads `Input.robot` (from whatever `RobotLibrary` is configured for the request).
- RoboServer verifies that the robot has a variable named `userLogin` and that this variable is marked as input.
- RoboServer now verifies that the attributes we have configured using `setAttribute` are compatible with the type of variable `userLogin`. As a result the type must have attributes named username and password and that these must both be text-based attributes (the next section describes the mapping between API and Design Studio attributes).
- If all input variables are compatible, RoboServer will start executing the robot.

If a robot requires multiple input variables, you must create all of them to execute the robot. You only have to configure required attributes; any no-required attributes that you do not configure through the API will just have a null value. If you have a robot that requires login to both Facebook and Twitter, you could define the input as follows.

```
Request request = new Request("Library:/Input.robot");
request.CreateInputVariable("facebook").SetAttributeEntry
  ("username", "scott").SetAttributeEntry("password", "facebook123");
request.CreateInputVariable("twitter").SetAttributeEntry
  ("username", "scott").SetAttributeEntry("password", "twitter123");
```

Attribute Types

When you define a new type in Design Studio, you select an attribute type for each attribute. Some attributes can contain text, like Short text, Long Text, Password, HTML, XML, and when used inside a robot, there may be requirements to the text stored in these attributes. If you store text in a XML attribute, the text must be a valid XML document. This validation occurs when the type is used inside a robot, but since the API does not know anything about the type, it does not validate attribute values in the same manner. As a result the API only has 8 attribute types versus the 19 available in Design Studio. This table shows the mapping between the API and Design Studio attribute types.

API to Design Studio mapping

API Attribute Type	Design Studio Attribute Type
Text	Short Text, Long Text, Password, HTML, XML, Properties, Language, Country, Currency, Refind Key
Integer	Integer
Boolean	Boolean
Number	Number
Character	Character
Date	Date
Session	Session
Binary	Binary, Image, PDF

The API attribute types are then mapped to .NET in the following way.

.Net Types for Attributes

API Attribute Type	Java Class
Text	`System.String` (string)
Integer	`System.Int64`
Boolean	`System.Boolean` (bool)
Number	`System.Double` (double)
Character	`System.Char` (char)
Date	`System.DateTime`
Session	`Com.Kapowtech.Robosuite.Api.Construct.Session`
Binary	`Com.Kapowtech.Robosuite.Api.Construct.Binary`

The `RqlObjectBuilder setAttribute` method is overloaded so you do not need to specify the attribute type explicitly when configuring an attribute through the API, as long as the right .NET class is used as an argument. Here is an example that shows how to set the attributes for an object with all possible (Design Studio) attribute types.

Recommended usage of `setAttribute`:

```
RqlObjectBuilder inputBuilder = request.CreateInputVariable("AllTypes");
inputBuilder.SetAttributeEntry("anInt", 42L);
inputBuilder.SetAttributeEntry("aNumber", 12.34d);
inputBuilder.SetAttributeEntry("aBoolean", true);
inputBuilder.SetAttributeEntry("aCharacter", 'c');
inputBuilder.SetAttributeEntry("aShortText", "some text");
inputBuilder.SetAttributeEntry("aLongText", "a longer text");
inputBuilder.SetAttributeEntry("aPassword", "secret");
inputBuilder.SetAttributeEntry("aHTML", "<html>bla</html>");
inputBuilder.SetAttributeEntry("anXML", "<tag>text</tag>");
inputBuilder.SetAttributeEntry("aDate", DateTime.Now);
inputBuilder.SetAttributeEntry("aBinary", (Binary) null);
inputBuilder.SetAttributeEntry("aPDF", (Binary)null);
inputBuilder.SetAttributeEntry("anImage", (Binary)null);
inputBuilder.SetAttributeEntry("aProperties", "name=value\nname2=value2");
inputBuilder.SetAttributeEntry("aSession", (Session)null);
inputBuilder.SetAttributeEntry("aCurrency", "USD");
inputBuilder.SetAttributeEntry("aCountry", "US");
inputBuilder.SetAttributeEntry("aLanguage", "en");
inputBuilder.SetAttributeEntry("aRefindKey", "Never use this as input");
```

In the preceding example we have to cast null values, because the C# compiler cannot otherwise determine which of the overloaded version of `SetAttributeEntry` method we want to call. However since unconfigured attributes will automatically be null, you never need to set null explicitly.

It is possible to specify the `Attribute` and `AttributeType` explicitly when creating input using the API. This approach is not recommended, but may be needed in rare cases, and would look like this.

Not recommended usage of `setAttribute`

```
RqlObjectBuilder inputBuilder = request.CreateInputVariable("alltypes");
inputBuilder.SetAttributeEntry(new AttributeEntry("anInt", "42",
  AttributeEntryType.Integer));
```

```
inputBuilder.SetAttributeEntry(new AttributeEntry("aNumber", "12.34",
  AttributeEntryType.Number));
inputBuilder.SetAttributeEntry(new AttributeEntry("aBoolean", "true",
  AttributeEntryType.Boolean));
inputBuilder.SetAttributeEntry(new AttributeEntry("aCharacter", "c",
  AttributeEntryType.Character));
inputBuilder.SetAttributeEntry(new AttributeEntry("aShortText", "some text",
  AttributeEntryType.Text));
inputBuilder.SetAttributeEntry(new AttributeEntry("aLongText", "a longer text",
  AttributeEntryType.Text));
inputBuilder.SetAttributeEntry(new AttributeEntry("aPassword", "secret",
  AttributeEntryType.Text));
inputBuilder.SetAttributeEntry(new AttributeEntry("aHTML", "<html>bla</html>",
  AttributeEntryType.Text));
inputBuilder.SetAttributeEntry(new AttributeEntry("anXML", "<tag>text</tag>",
  AttributeEntryType.Text));
inputBuilder.SetAttributeEntry(new AttributeEntry("aDate",
  "2012-01-15 23:59:59.123", AttributeEntryType.Date));

inputBuilder.SetAttributeEntry(new AttributeEntry("aBinary", null,
  AttributeEntryType.Binary));
inputBuilder.SetAttributeEntry(new AttributeEntry("aPDF", null,
  AttributeEntryType.Binary));
inputBuilder.SetAttributeEntry(new AttributeEntry("anImage", null,
  AttributeEntryType.Binary));
inputBuilder.SetAttributeEntry(new AttributeEntry("aProperties",
  "name=value\nname2=value2", AttributeEntryType.Text));
inputBuilder.SetAttributeEntry(new AttributeEntry("aCurrency", "USD",
  AttributeEntryType.Text));
inputBuilder.SetAttributeEntry(new AttributeEntry("aCountry", "US",
  AttributeEntryType.Text));
inputBuilder.SetAttributeEntry(new AttributeEntry("aLanguage", "en",
  AttributeEntryType.Text));
inputBuilder.SetAttributeEntry(new AttributeEntry("aRefindKey",
  "Never use this as input", AttributeEntryType.Text));
```

All attribute values must be provided in the form of strings. The string values are then converted to the appropriate .NET objects based on the `AttributeEntryType` provided. This is only useful if you build other generic APIs on top of the Kapow .NET API.

Execution Parameters

In addition to the `CreateInputVariable` method, the Request contains a number of properties that controls how the robot executes on a RoboServer.

Execution Control Methods on Request

MaxExecutionTime	Controls the execution time (in seconds) of the robot. When this time has elapsed, the robot will be stopped by RoboServer. The timer does not start until the robot begins to execute, so if the robot is queued on RoboServer, this is not taken into account.
StopOnConnectionLost	When true (default) the robot will stop if RoboServer discovers that the connection to the client application has been lost. You should have a very good reason for setting this value to false; if your code is not written to handle this, your application will not perform as expected.

`StopRobotOnApiException`	When true (default) the robot is stopped by RoboServer after the first API exception is raised. By default most steps in a Robot will raise an API exception if the step fails to execute. - Configure this value on the Steps Error Handling tab. When set to false, the robot will continue to execute regardless of API exceptions. However, unless your application is using the `IRobotResponseHandler` for streaming the results, an exception is still thrown by Execute(), so be extremely cautious when setting this to false.
`Username, Password`	Sets the RoboServer credentials. RoboServer can be configured to require authentication. When this option is enabled, the client must provide credentials or RoboServer will reject the request.
`RobotLibrary`	Assigns a `RobotLibrary` to the request. A robot library instructs RoboServer where to find the robot identified in the request. For more examples related to the various library types and their usage, see Robot Libraries.
`ExecutionId`	Allows you to set the `executionId` for this request. If you don't provide one, RoboServer will generate one automatically. The execution ID is used for logging, and also needed to stop the robot programmatically. The ID must be globally unique (over time). If two robots use the same execution ID, the logs will be inconsistent. Setting this is useful if your robots are part of a larger workflow and you already have a unique identifier in your client application, because it allows you to join the robot logs with the rest of the system.
`setProject(String)`	This method is used solely for logging purposes. The Management Console uses this field to link log messaged to project, so the log views can filter by project. If your application is not using the `RepositoryRobotLibrary` you should probably set this value to inform the RoboServer logging system which project (if any) this robot belongs to.

Robot Libraries

In Design Studio robots are grouped into projects. If you look in the file system, you will see that these projects are represented by a folder with the only constraint that it must contain a folder named `Library`.

When you build the execute request for RoboServer, you identify the robot by a robot URL:

```
Request request = new Request("Library:/Input.robot");
```

Here, `Library:/` is a symbolic reference to a robot library, in which the RoboServer should look for the robot. The `RobotLibrary` is then specified on the builder like this:

```
request.setRobotLibrary(new DefaultRobotLibrary());
```

There are three different robot library implementations, which one to select depends on you deployment environment.

Robot Libraries

Library Type	Description
DefaultRobotLibrary	This library configures RoboServer to look for the robot in the current project folder. This folder is defined in the Settings application.
	If you have multiple RoboServers you must deploy your robots on all RoboServers.
	This robot library is not cached, so the robot is reloaded from disk with every execution. This approach makes the library usable in a development environment where robots change often, but not suitable for a production environment.
EmbeddedFileBasedRobotLibrary	This library is embedded in the execute request sent to RoboServer. To create this library you must create a zip file containing the robots and all its dependencies (types, snippets and resources). This can be done the **Tools** > **Create Robot Library File** menu in Design Studio.
	The library is sent with every request, which adds some overhead for large libraries, but the libraries are cached on RoboServer, which offers best possible performance.
	One strength is that robots and code can be deployed as a single unit, which offers clean migration from QA environment to production environment. However, if the robots change often you will have to redeploy them often.
	You can use the following code to configure the embedded robot library for your request.
	``` var request = new Request ("Library:/Tutorials/NewsMagazine. robot"); var stream = new FileStream ("c:\\embeddedLibrary.robotlib", FileMode.Open); request.RobotLibrary = new EmbeddedFileBasedRobotLibrary (stream); ```

Library Type	Description
`RepositoryRobotLibrary`	This is the most flexible `RobotLibrary`. This library uses the Management Console's built-in repository as a robot library. When you use this library, RoboServer will contact the Management Console which will send a robot library containing the robot and its dependencies. Caching occurs on a per robot basis, inside both Management Console and RoboServer. Inside Management Console, the generated library is cached based on the robot and its dependencies. On RoboServer, the cache is based on a timeout, so it does not have to ask the Management Console for each request. In addition, the library loading between RoboServer and Management Console uses HTTP public/private caching, to further reduce bandwidth. If `NewsMagazine.robot` is uploaded to the Management Console, you can use the repository robot library when executing the robot: ```\nvar request = new Request\n ("Library:/Tutorials/NewsMagazine.\n robot");\n request.RobotLibrary =\n new RepositoryRobotLibrary\n ("http://localhost:50080",\n "Default Project", 60000);\n``` This will instruct RoboServer to load the robot from a local Management Console and cache it for one minute before checking with the Management Console to see if a new version of the robot (it's type and snippets) is available. In addition, any resource loaded through the `Library:/ protocol`, will cause RoboServer request the resource directly from the Management Console.

.NET Advanced

This section describes advanced API features, including output streaming, logging and SSL configuration, as well as parallel execution.

Load Distribution

Inside the `RequestExecutor`, the executor is given an array of RoboServers. As the executor is constructed, it tries to connect to each RoboServer. Once connected, it sends a ping request to each RoboServer to discover how the server is configured.

Load balanced executor

```
RoboServer prod = new RoboServer("prod.kapow.local", 50000);
```

```
RoboServer prod2 = new RoboServer("prod2.kapow.local", 50000);
Cluster cluster = new Cluster("Prod", new RoboServer[]{ prod, prod2}, false);
Request.RegisterCluster(cluster);
```

Load is distributed to each online RoboServer in the cluster, based on the number of unused execution slots on the RoboServer. The next request is always distributed to the RoboServer with the most available slots. The number of available execution slots is obtained through the initial Ping response, and the executor keeps track of each robot it starts, and when it completes. The number of execution slots on a RoboServer is determined by the **max concurrent robots** setting on the Servers tab.

If a RoboServer goes offline, it will not receive any robot execution requests before it has successfully responded to the ping request.

Two Client Rule

You should only have one API client using a given cluster of RoboServer. If you have multiple .NET applications running robots against the same RoboServers, this will result in reduced performance.

Data Streaming

Sometimes you need to present the results from a robot execution in real-time. In such cases, you want the API to return the extracted values immediately instead of waiting for the robot to finish its execution and access the `RqlResult`.

The API offers the possibility to receive a callback every time the API receives a value that was returned by the Robot. Do this through the `IRobotResponseHandler` interface.

Response streaming usingAbstractFailFastRobotResponseHandler

```
using System;
using Com.KapowTech.RoboSuite.Api;
using Com.KapowTech.RoboSuite.Api.Repository.Construct;
using Com.KapowTech.RoboSuite.Api.Construct;
using System.IO;
using Com.KapowTech.RoboSuite.Api.Engine.Hotstandby;

namespace Examples
{

    public class DataStreaming {

        public static void Main(String[] args)  {

            var server = new RoboServer("localhost", 50000);
            var cluster = new Cluster("MyCluster", new RoboServer[] { server },
                false);
            Request.RegisterCluster(cluster);

            var request = new Request("Library:/Tutorials/NewsMagazine.robot");
            IRobotResponseHandler handler = new SampleResponseHandler();
            request.Execute("MyCluster", handler);
        }

    }


    public class SampleResponseHandler : AbstractFailFastRobotResponseHandler
```

```
    {
        override public void HandleReturnedValue(RobotOutputObjectResponse
          response, IStoppable stoppable)
        {
            var title = response.OutputObject["title"];
            var preview = response.OutputObject["preview"];
            Console.WriteLine(title + ", " + preview);
        }
    }
}
```

The preceding example uses the second execute method of the Request, which expects a
`RobotResponseHandler` in addition to the name of the cluster to execute the robot on. In this example,
create a `IRobotResponseHandler` by extending `AbstractFailFastRobotResponseHandler`,
which provides default error handling, to handle the values returned by the robot.

The `handleReturnedValue` method is called whenever the API receives a returned value from
RoboServer. The `AbstractFailFastRobotResponseHandler` used in this example, will throw
exceptions in the same way as the non-streaming execute method. This means that an exception is
thrown in response to any API exceptions generated by the robot.

The `IRobotResponseHandler` has several methods which can be grouped into three categories.

Robot life cycle events

Methods called when the robot's execution state change on RoboServer, such as when it starts and
finishes its execution.

Robot data events

Methods which are called when the robot returns data or errors to the API.

Additional error handling

Methods which are called either due to an error inside RoboServer or in the API.

`RobotResponseHandler` **- robot life cycle events**

Method name	Description
void requestSent(RoboServer roboServer, ExecuteRequest request)	Called when the `RequestExecutor` finds the server which will execute the request.
void requestAccepted(String executionId)	Called when the found RoboServer has accepts the request and puts it into it queue.
void RobotStarted(IStoppable stoppable)	Called when the RoboServer begins to execute the robot. This usually occurs immediately after the robot is queued, unless the RoboServer is under heavy load, or used by multiple API clients.
void robotDone(RobotDoneEvent reason)	Called when the robot is done executing on RoboServer. The `RobotDoneEvent` is used to specify if the execution terminated normally, due to an error, or if it was stopped.

`RobotResponseHandler` **- robot data events**

Method name	Description
`void HandleReturnedValue(RobotOutputObjectResponse response, IStoppable stoppable)`	Called when the robot is executed a Return Value action and the value has been returned via the socket to the API.
`void HandleRobotError(RobotErrorResponse response, IStoppable stoppable)`	Called when the robot raises an API exception. Under normal circumstances the robot stops executing after the first API exception. This behavior can be overridden by using `Request.StopRobotOnApiException = false`, in which case this method is called multiple times. This is useful if you want a data streaming robot to continue to execute regardless of any generated errors.
`void HandleWriteLog(RobotMessageResponse response, IStoppable stoppable)`	Called if the robot executes the Write Log action. This is useful to provide additional logging info from a robot.

`RobotResponseHandler` **- additional error handling**

Method name	Description
`void HandleServerError(ServerErrorResponse response, IStoppable stoppable)`	Called if RoboServer generates an error, for instance if the server is too busy to process any requests, or if an error occurs inside RoboServer, which prevents it from starting the robot.
`void handleError(RQLException e, IStoppable stoppable)`	Called if an error occurs inside the API. Most commonly if the client loses the connection to RoboServer.

Many of the methods include a `IStoppable` object, which can be used to stop for instance in response to a specific error or value returned.

Some of these methods allow you to throw an `RQLException`, if you do this you should be aware of the consequences. The thread that calls the handler is the thread that calls `Request.Execute()`, this means that any exceptions thrown will bubble up the call stack and out the execute method. If you throw an exception in response to `handleReturnedValue`, `handleRobotError` or `handleWriteLog` it is your responsibility to invoke Stoppable.stop(), or the robot may continue to execute even though the call to `Request.Execute()` has completed.

Data streaming is most often used in one of the following use cases.
- Ajax based web application, where results are presented to the user in real-time. If data is not streamed, results cannot be shown until the robot is done running.
- Robots that return so much data that the client would not be able to hold it all in memory throughout the robot's execution.
- Processes that need to be optimized so the extracted values are processed in parallel with the robot execution.
- Processes that store data in databases in a custom format.
- Robots that should ignore or require custom handling of API exceptions (see the following).

Response and error collecting using AbstractFailFastRobotResponseHandler:

```
using System;
```

```
using System.Collections;
using System.Collections.Generic;
using Com.KapowTech.RoboSuite.Api;
using Com.KapowTech.RoboSuite.Api.Repository.Construct;
using Com.KapowTech.RoboSuite.Api.Construct;
using System.IO;
using Com.KapowTech.RoboSuite.Api.Engine.Hotstandby.Interfaces;

namespace Examples
{
    public class DataStreaming
    {

        public static void Main(String[] args)
        {

            var server = new RoboServer("localhost", 50000);
            var cluster = new Cluster("MyCluster", new RoboServer[] { server },
              false);
            Request.RegisterCluster(cluster);

            var request = new Request("Library:/Tutorials/NewsMagazine.robot");
            request.StopRobotOnApiException = false;  // IMPORTANT!!

            ErrorCollectingRobotResponseHandler handler =
              new ErrorCollectingRobotResponseHandler();
            request.Execute("MyCluster", handler); // blocks until robot is
              done, or handler throws an exception

            Console.WriteLine("Extracted values:");
                foreach (RobotOutputObjectResponse response in handler.
                 GetOutput())
            {
                var title = response.OutputObject["title"];
                var preview = response.OutputObject["preview"];
                Console.WriteLine(title + ", " + preview);
            }

            Console.WriteLine("Errors:");
            foreach (RobotErrorResponse error in handler.GetErrors())
            {
                Console.WriteLine(error.ErrorLocationCode + ", " + error.
                  ErrorMessage);
            }
        }
    }


    public class ErrorCollectingRobotResponseHandler :
      AbstractFailFastRobotResponseHandler {

        private IList<RobotErrorResponse> _errors =
          new List<RobotErrorResponse>();
        private IList<RobotOutputObjectResponse> _output =
          new List<RobotOutputObjectResponse>();

        override public void HandleReturnedValue(RobotOutputObjectResponse
          response, IStoppable stoppable)  {
            _output.Add(response);
        }


        override public void HandleRobotError(RobotErrorResponse response,
          IStoppable stoppable) {
```

```
            // do not call super as this will stop the robot
            _errors.Add(response);
        }

        public IList<RobotErrorResponse> GetErrors() {
            return _errors;
        }

        public IList<RobotOutputObjectResponse> GetOutput() {
            return _output;
        }
    }
}
```

The preceding example shows how to use a `IRobotResponseHandler` that collects returned values and errors. This type of handler is useful if the robot should continue to execute even when error are encountered, which can be useful if the website is unstable and occasionally times out. Notice that only robot errors (API exceptions) are collected by the handler. If the connection to RoboServer is lost, `Request.Execute()` will still throw an `RQLException` (and the robot will be stopped by RoboServer).

For more details, check the `IRobotResponseHandler` documentation in the `API\robosuite-dotnet-api\docs` folder in the Kapow installation folder.

SSL

The API communicates with RoboServer through an `RQLService`, which is a RoboServer component that listens for API requests on a specific network port. When you start a RoboServer, you specify if it should use the encrypted SSL service, or the plain socket service, or both (using two different ports). All RoboServers in a cluster must be running the same `RQLService` (although the port may be different).

Assuming we have started a RoboServer with the SSL `RQLService` on port 50043:

```
RoboServer -service ssl:50043
```

We can use the following code:

```
RoboServer server = new RoboServer("localhost", 50043);
boolean ssl = true;
Cluster cluster = new Cluster("MyCluster", new RoboServer[] {server}, ssl);
Request.RegisterCluster(cluster);
```

All we need to do is to create the cluster as an SSL cluster and specify the SSL port used by each RoboServer. Now all communication between RoboServer and the API will be encrypted.

In addition to data encryption, SSL offers the possibility to verify the identity of the remote party. This type of verification is very important on the Internet, as rogue Web sites could otherwise pretend to be someone they are not. Most often your API client and RoboServers are on the same local network, so you rarely need to verify the identity of the other party, but the API supports this feature should it become necessary.

See Examples to find out how to compile and run the included SSL example.

Repository Integration

In the Management Console you also specify clusters of RoboServers, which are used to execute scheduled robots, as well as robots executed as REST services. The API allows you to use the `RepositoryClient` to obtain cluster information from Management Console. See the `RepositoryClient` documentation for details.

Repository Integration

```
using System;
 using Com.KapowTech.RoboSuite.Api;
 using Com.KapowTech.RoboSuite.Api.Construct;
 using Com.KapowTech.RoboSuite.Api.Repository.Engine;

 namespace Examples
 {
     public class RepositoryIntegration
     {
         public static void Main(String[] args)
         {
             string userName = "admin";
             string password = "admin";
             RepositoryClient client = new RepositoryClient
                ("http://localhost:50080", userName, password);

             Request.RegisterCluster(client, "Production");
             var request = new Request("Library:/Tutorials/NewsMagazine.robot");
             var result = request.Execute("Production");
             Console.WriteLine(result.ToString());
         }
     }
 }
```

The preceding example shows how to create a `RepositoryClient` that connects to a Management Console deployed on localhost port 50080.

If the Management Console requires authentication, you must pass a username and password, otherwise you may pass null for both. When we register the `RepositoryClient`, we specify the name of a cluster that exists on the Management Console. This will then query the Management Console to get a list of RoboServers configured for this cluster, and check every two minutes to see if the cluster configuration has been updated on the Management Console.

This integration allows you to create a cluster on Management Console that you can change dynamically using the Management Console user interface. When you use a Management Console cluster with the API usage should be exclusive, and you should not use it for scheduling robot, as this would break the two client rule.

Executor Logger

When you execute a request, the execute method throws an exception if a robot generates an error. Other types of errors and warnings are reported through the `ExecutorLogger` interface. In the previous examples, we have not provided any `ExecutionLogger` when executing robots, which means we get the default implementation that will write to system out. Let's see how the `ExecutorLogger` reports if one of our RoboServers goes offline.

The example configures a cluster with a RoboServer which is not online.

ExecutorLogger, offline server example:

```
RoboServer rs = new RoboServer("localhost", 50000);
Cluster cluster = new Cluster("name", new RoboServer[]{rs}, false);
Request.RegisterCluster(cluster);
```

If you run this example it should print the following to the console.

ExecutorLogger, offline RoboServer console output:

```
RoboServer[Host=localhost, Port=50000]' went offline.
Com.KapowTech.RoboSuite.Api.Engine.UnableToConnectException:...........
```

Often you don't want to have your application writing directly to `System.out`, in that case you can provide a different `IExecutorLogger` implementation; you can do so when registering the cluster:

Use DebugExecutorLogger:

```
Request.RegisterCluster(cluster, new DebugExecutorLogger());
```

This example uses the `DebugExecutorLogger()` that also prints to `System.out`, but only if the API debugging is enabled. Alternatively you can provide your own implementation of the `ExecutorLogger` to control how error messages should be handled.

Under the Hood

This section will explain what is going on under the hood when you register a cluster and execute requests.

When you register a cluster with the request, a `RequestExecutor` is created behind the scene. This `RequestExecutor` is stored in a Map using the cluster name as key. When a request is executed, the provided cluster name is used to find the associated `RequestExecutor` and execute the request.

Lets look at a short example.

Normal Execution

```
 public static void Main(String[] args)
{
    RoboServer server = new RoboServer("localhost", 50000);
    Cluster cluster = new Cluster("MyCluster", new RoboServer[]{ server}, false);
    Request.RegisterCluster(cluster);

    var request = new Request("Library:/Tutorials/NewsMagazine.robot");
    request.RobotLibrary = new DefaultRobotLibrary();
    var result = request.Execute("MyCluster");
    Console.WriteLine(result);
}
```

Now write the same example by using the `hiddenRequestExecutor` directly.

Under the hood execution:

```
 public static void Main(String[] args)
{
    RoboServer server = new RoboServer("localhost", 50000);
    Cluster cluster = new Cluster("MyCluster", new RoboServer[]{ server}, false);
    RequestExecutor executor = new RequestExecutor(cluster);

    var request = new Request("Library:/Tutorials/NewsMagazine.robot");
    request.RobotLibrary = new DefaultRobotLibrary();
    var result = executor.Execute(request);
    Console.WriteLine(result);
}
```

The `RequestExecutor` is hidden by default, so you do not have to keep track of it. You may only create one `RequestExecutor` per cluster, so if you use it directly you need to store a reference to it throughout your application. Using `Request.RegisterCluster(cluster)` means that you can ignore the `RequestExecutor` and lifecycle rules.

The `RequestExecutor` contains the necessary state and logic which provides the load balancing and failover features. Using the `RequestExecutor` directly also offers a few extra features.

Request Executor Features

When the `RequestExecutor` is not connected to a repository, you can dynamically add or remove RoboServers, by calling `AddRoboServer(..)` and `RemoveRoboServer(..)`. These methods modifies the distribution list used inside the `RequestExecutor`.

`RequestExecutor.TotalAvailableSlots` property contains the number of unused execution slots across all RoboServers in the internal distribution list.

By using these methods, you can dynamically add RoboServers to your `RequestExecutor` once the number of available execution slots becomes low.

When you create the `RequestExecutor`, you may optionally provide an `IRqlEngineFactory`. The `IRqlEngineFactory` allows you to customize which `RQLProtocol` is used when connecting to a `RoboServer`. This is only needed under very rare circumstances, for instance if you want use a client certificate to increase security. See API Client Certificates for details.

Repository API

The Repository API allows you to query the Management Console's repository to get a list of projects, robots and the input required to call a robot. It also allows you to programmatically deploy robots, types and resource files.

Repository Client

Communication with the repository is achieved through the `RepositoryClient` in the namespace `Com.KapowTech.RoboSuite.Api.Repository.Engine`.

Let's look at an example .

Get Projects from Repository

```
string UserName = "admin";
string Password = "admin1234";
RepositoryClient client = new RepositoryClient("http://localhost:50080/", UserName,
Password);
Project[] projects = client.GetProjects();
    foreach(Project p in projects) {
Console.WriteLine(p);
}
```

Here we see a `RepositoryClient` configured to connect to Management Console's repository on `http://localhost:50080/`, with a username and password. If the Management Console is not password protected, you must supply null for user name and password.

Once the `RepositoryClient` is created, we use the `GetProjects()` method to query the repository for a list of projects. Notice that when calling any of the `RepositoryClient` methods, a `RepositoryClientException` is thrown if an error occurs.

The `RepositoryClient` has the following eleven methods.

Methods of the RepositoryClient:

Method signature	Description
`void DeleteResource(string projectName, string resourceName, boolean silent`	Deletes a resource from a project. The `resourceName` argument uses the full path of the resource.
`void DeleteRobot(string projectName, string robotName, boolean silent)`	Deletes a robot from a project. The `robotName` argument uses the full path of the robot.
`void DeleteType(string projectName, string typeName, boolean silent)`	Deletes a type from a project. The `typeName` argument uses the full path of the type.
`void DeleteSnippet(string projectName, string snippetName, boolean silent)`	Deletes a snippet from a project. The `snippetName` argument uses the full path of the snippet.
`void DeployLibrary(string projectName, EmbeddedFileBasedRobotLibrary library, boolean failIfExists)`	Deploys a library to the server. Robots, types and resources are overridden unless `failIfExists` is true.
`void DeployResource(string projectName, string resourceName, byte[] resourceBytes, boolean failIfExists)`	Deploys a resource to a project. If a resource with the given name already exists it can be overridden by setting `failIfExists` to false. The `resourceName` argument uses the full path of the resource.
`void DeployRobot(string projectName, string robotName, byte[] robotBytes, boolean failIfExists)`	Deploys a robot to a project. If a robot with the given name already exists it can be overridden by setting `failIfExists` to false. The `robotName` argument uses the full path of the robot.
`void DeployType(string projectName, string typeName, byte[] typeBytes, boolean failIfExists)`	Deploys a type to a project. If a type with the given name already exists it can be overridden by setting `failIfExists` to false. The `typeName` argument uses the full path of the resource.
`void DeploySnippet(string projectName, string snippetName, byte[] snippetBytes, boolean failIfExists)`	Deploys a snippet to a project. If a snippet with the given name already exists it can be overridden by setting `failIfExists` to false. The `snippetName` argument uses the full path of the snippet.
`Project[] GetProjects()`	Returns the projects that exist in this repository.

Method signature	Description
`Cluster[] GetRoboServerClusters()`	Returns a list of clusters and online (valid) RoboServers that are registered with the Management Console running the repository.
`Cluster[] GetRoboServerClusters(boolean onlineRoboServer)`	Returns a list of clusters and RoboServers that are registered with the Management Console. Use the `onlineRoboServer` flag to indicate if the clusters should include only RoboSevers that the are online or all of the RoboServers.
`Cluster AddRoboServer(String clusterName, int portNumber, String host)`	Adds a new RoboServer to a cluster.
`Robot[] GetRobotsInProject(String projectName)`	Returns the full paths of robots available in the project.
`RobotSignature GetRobotSignature(String projectName, String robotName)`	Returns the robot signature with the full path of the robot, as well as the input variables required to execute this robot and a list of the types it may return or store.
`RepositoryFolder GetProjectInventory(String projectName)`	Returns the entire tree of folders and files from the repository.
`RepositoryFolder GetFolderInventory(String projectName, String folderPath)`	Returns the folders and files of the subfolder in the specified project from the repository.
`RepositoryFolder GetFileInventory(String projectName, String folderPath, String fileName, RepositoryFile.Type fileType)`	Gets the file and the referenced files from the management console. Note that the file inventory is wrapped in a RepositoryFolder to get references.
`Void DeleteFile(RepositoryFile file, bool silent)`	Deletes the specified file from the repository.
`Date GetCurrentDate()`	Returns current date and time of the Management Console.
`byte[] GetBytes(RepositoryFile file)`	Returns the size in bytes of the specified file in the repository.
`ComputeChecksum(byte[] bytes)`	Returns the checksum of the specified file to verify data integrity.
`void UpdateFile(RepositoryFile file, byte[] bytes)`	Updates the specified file in the repository with new bytes.
`void MoveFile(RepositoryFile sourceFile, String destFolderPath)`	Moves the specified file from the repository to a folder specified in `destFolderPath`.
`void RenameRobot(RepositoryFile robotFile, String newName)`	Renames the specified robot file.
`void DeleteFolder(String projectName, String folderPath)`	Deletes the specified folder in the repository.
`void DeleteRoboServer(String clusterName, RoboServer roboServer)`	Deletes a RoboServer.

Method signature	Description
`Map<String, String> getInfo()`	Returns information about the Management Console and the Repository API The method returns a mapping of the following: • "application" to the version of the Management Console containing major, minor and dot version, for example, `10.3.2` • "repository" to the ID of the latest DTD used by the Repository API, such as: `//Kapow Technologies//DTD Repository 1.5//EN` • "rql" to the ID of the latest DTD used by the Robot Query Language API, such as: `//Kapow Technologies//DTD RoboSuite Robot Query Language 1.13//EN`

Note The full path is relative to your project folder.

Check the .Net documentation for details. The .Net documentation is located inside you Kofax Kapow installation at `/API/robosuite-dotnet-api/docs/RoboSuite .NET API.chm`.

If authentication is enabled on the repository, the request may be declined if the credentials given do not have sufficient access.

The repository is accessed via http. When using the .Net version of the Repository API, any proxy servers configured for Internet Explorer will be used by the Repository API.

Deployment via Repository Client

The following example shows how to deploy a robot and a type from the local file system using the `RepositoryClient`.

Deploying to Repository
```
string user = "test";
string password = "test1234";
RepositoryClient client = new RepositoryClient("http://localhost:50080", user,
  password);

byte[] robotBytes = File.ReadAllBytes("c:\\MyRobots\\Library\\Test.robot");
byte[] typeBytes = File.ReadAllBytes("c:\\MyRobots\\Library\\Test.type");

// we assume that no one has deleted the Default project
client.deployRobot("Default project", "Test.robot", robotBytes, true);
client.deployType("Default project", "Test.type", typeBytes, true);
```

Repository API as Rest

The repository can also be accessed via restful services.

Examples

The Kofax Kapow installation contains six additional API code examples, which are located in `API\robosuite-dotnet-api\example`.

After completing the configuration steps, both the server and client will be configured to use SSL. Running `RunSslRobot.exe` can be used to verify the configuration.

Compiling & Running the Examples

To compile the examples, run `build.bat` from a command prompt. This will create six .exe files that can be run directly.

The .exe files rely on `robosuite-dotnet-api.dll` and `log4net.dll` both of which are located in the examples directory. Both files are identical copies of the ones located in the bin folder and are copied to this folder to make it easier to run the examples.

Each example program prints a small usage text when run without any arguments.

C# Compiler Issues

The `build.bat` file assumes that the C# compiler is available in the path.

.NET Framework 4.0

The API and accompanying log4net is built targeting the .NET framework 4.0 client profile. For details on the .NET framework 4.0 client profile, see `http://msdn.microsoft.com/en-us/library/cc656912.aspx`.

SSL Example

To run the SSL example `RunSslRobot.exe`, the RoboServer must be configured to use SSL and the certificate has to be imported on the client machine. This topic shows you how to configure SSL using a self-signed certificate on a windows PC running a local RoboServer.

Configure the RoboServer

1. On the computer running a RoboServer, start the RoboServer Settings application located in **Start** > **All Programs** > **Kapow**.

2. In the `Settings` application go to the Certificates tab

3. Click **Change** under **API** and select the file `API\robosuite-dotnet-api\example\server.pfx`

4. When prompted for a password type 123.

Configure the API Client

1. Run the command `mmc.exe` in the command line.

2. On the Console menu, click **Add/Remove Snap-in**.

3. Under Snap-in, double-click **Certificates**, and select to manage certificates for the local computer and click **Finish**.

4. When the certificates snap-in loads, expand the node **Certificates -> Trusted root Certification Authorities**, right click the **Certificates** node and click the menu item **All tasks -> Import**.

   This will start the Certificate Import Wizard. When prompted for the certificate file, browse to `API \robosuite-dotnet-api\example\server.pub.cer` and complete the import.

Chapter 3

Kapow Control Protocol

Kapow Control Protocol or KCP is an execution protocol for executing robots over Java Message Service (JMS), using Google Protocol Buffers (Protobuf).

The KCP protocol defines a set of messages that enable you to communicate with a RoboServer. The following messages are defined.

Message	Direction	Notes	Queue/Topic
Message	both	A container, wrapping all the following messages.	All
ExecuteRobot	sending	Includes a robot URL for the RoboServer to get the robot from the repository.	Execute
StopRobot	sending	Sent to interrupt a running robot.	Control
RobotEvent	receiving	(START_REQUESTED STARTED, STOP_REQUESTED, STOPPED, FAILED; ENDED)	Result
ServerMessage	receiving	Either info or error from the server regarding a run.	Result
RobotResult	receiving	A Robot Result is send every time a return value step is executed in the robot.	Result
RobotRunStatus	receiving	Summary of run including the number of returned RobotResult messages.	Result

KCP communicates over three so called JMS Destinations listed in the following table.

Name	Destination type	Description
Execute	Queue	Messages to a RoboServer
Result	Queue	Information from RoboServers
Control	Topic	Broadcast to all RoboServers

The following is an example of a normal KCP life cycle.

1. An ExecuteRobot message is sent. When the message is picked up by a RoboServer, it sends a RobotEvent (START_REQUESTED) that informs you which RoboServer is handling the execution.

2. START_REQUESTED is followed by a RobotEvent (STARTED). During the execution, the robot might send multiple RobotResults that you can pick up from the result queue.

3. When the robot stops, it sends a RobotEvent (ENDED) and a RobotRunStatus that informs you about how many results were returned.

Build a JMS Client

To use KCP you must set up the following components.

- Kapow JMS client that includes the following components:
  - Management Console
  - RoboServer
  - JMS broker
- A JMS client api In your language (see http://activemq.apache.org for more information)
- The protocol definition file (`kcp.proto`).
- The Protocol Buffers compiler version 3 (proto3) that you can download from https://github.com/google/protobuf/releases (see https://developers.google.com for more information).
- The latest version of the `Protobuf.jar` file.

In the following tutorials we use Java, and ActiveMQ Client to connect to the JMS broker.

- KCP Tutorial 1: Compile KCP, Connect to JMS Broker, and Send a Message
- KCP Tutorial 2: Consume Specific Results
- KCP Tutorial 3: Stop Robot Execution

KCP Tutorial 1: Compile KCP, Connect to JMS Broker, and Send a Message

In this tutorial we will compile KCP, connect to a JMS broker, and send a message. The resulting code can be found in the `Tutorial1.java` file.

Prerequisites

- Install Protobuf compiler.
- Use a programming language that supports Protobuf and JMS. In this tutorial we use Java.
- Set up the language dependent Protobuf library. In this tutorial: Java `protobuf.jar`.
- Set up and start ActiveMQ JMS message broker.

Step 1. Create the language dependent KCP definition

From the command line run the compiler with the following parameters:

```
protoc --java_out=[DestinationFolder] kcp.proto
```

The above command creates `com.kapowtech.kcp` Java package structure in the destination folder with a single file called `Kcp.java`. Do not change this file, it is simply a helper to create the Protobuf objects. The package must be included in your tutorial project.

Step 2. Connect to the broker

A broker can be configured to connect in many different ways, such as using credentials and certificates. In this example we assume a standard configuration of the broker, where anonymous access is allowed. To connect we only need a broker URI.

```
public void run() {
try {
//Create a ConnectionFactory
ActiveMQConnectionFactory
connectionFactory = new
ActiveMQConnectionFactory(BROKER_URI);
//Create a Connection
Connection connection = connectionFactory.createConnection();
connection.start();
//Create a Session
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

The code above connects to the broker and creates a session.

Step 3. Connect to the execution queue

On this step we connect to a queue to send a message. The queue name must include the same namespace and cluster as the RoboServer. The execution queue name consists of the following.

```
[NAMESPACE].KCP.[CLUSTER_NAME].Execute
```

For example, `Kapow.KCP.Production.Execute`.

```
private final String NAMESPACE = "Kapow" ; // Must match the namespace used by the
 RoboServer
private final String ENCODING = "KCP" ; // Must be KCP
private final String CLUSTER = "Production" ; // Must match the cluster used by the
 RoboServer
private final String EXECUTE = "Execute" ;
private final String EXECUTE_QUEUE = NAMESPACE + "." + ENCODING + "." + CLUSTER + "." +
EXECUTE ;
...
//Create the destination (Topic or Queue)
Destination destination = session.createQueue(EXECUTE_QUEUE);
```

The code above creates a queue if it does not exist.

When the queue is created, we add a producer and send the first message.

```
//Create a MessageProducer from the Session to the Topic or Queue
MessageProducer producer = session.createProducer(destination);
producer.setDeliveryMode(DeliveryMode. PERSISTENT );
//Create a message
Kcp.Message kcpMessage = createExecuteRobotMessage(); //we will get to this later
BytesMessage jmsMessage = session.createBytesMessage();
jmsMessage.writeBytes(kcpMessage.toByteArray());
jmsMessage.setStringProperty( "version" , "1" );
//Tell the producer to send the jms message
producer.send(jmsMessage);
```

When sending messages to Kapow using JMS, you must set a version property on the JMS message. The version is the version of the KCP message format and is currently 1.

```
jmsMessage.setStringProperty("version", "1");
```

Step 4. Create a KCP execute message

To create an `ExecuteRobot` message, use `ExecuteRobot` and `Message` classes generated from `kcp.proto` on Step 1. The message requires a robot url and a unique execution id. The URL must refer to the robot in a Management Console repository as in the following example.

```
http://[user]:[pass]@[host]:[port]/[MCName?project=[project name]&robot=[robotname]
```

Code example

```
private final String REPOSITORY = "http://admin:admin@localhost:8080/
ManagementConsole" ;
private final String PROJECT = "Default project" ;
private final String ROBOT = "MyTutorialRobot.robot" ;
private String executionId = UUID. randomUUID().toString(); // Must be unique across
 all clusters and time
...
// Create a RobotExecution message wrapped in a Message structure for sending.
private Kcp.Message createExecuteRobotMessage() {
Kcp.ExecuteRobot executeRobot = Kcp.ExecuteRobot. newBuilder()
.setRobotUrl( REPOSITORY + "?project=" + PROJECT + "&robot=" + ROBOT )
.setExecutionId( executionId )
// .setInput(createInputObjects()) //we will get to this in next step
.build();
return Kcp.Message. newBuilder()
.setExecuteRobot(executeRobot)
.build();
}
```

Refer to the code example for variables, boilerplate and context.

Step 5. Adding input objects

If you want to run robots using inputObjects, they need to be added to the KCP. In the following example we create a MyTutorialType named myTutorialObject with three attributes as an input object.

```
/**
*Create the test input object myTutorialObject of the type MyTutorialType
* @return an input object
*/
private Kcp.Structure createInputObjects() {
//create a map of 3 attributes for the myTutorialObject
Map<String, Kcp.Value> attributes = new HashMap<>();
attributes.put( "myInteger" , kcp.Value. newBuilder().setInteger( 42 ).build());
attributes.put( "myString" , Kcp.Value. newBuilder().setString( "" ).build());
attributes.put( "myDate" , Kcp.Value. newBuilder().setTimestamp(System.
 currentTimeMillis()).build());
//wrapping of attributes in structure
Kcp.Structure myTutorialObjectStructure = Kcp.Structure. newBuilder()
.putAllElements(attributes)
.build();
//Create a map of all the input objects in this case just a single object
Map<String, Kcp.Value> inputObjects= new HashMap<>();
//add myTutorialObject to the input object map.
inputObjects.put( "myTutorialObject" ,
Kcp.Value. newBuilder().setStructure(myTutorialObjectStructure).build());
Return Kcp.Structure. newBuilder()
.putAllElements(inputObjects)
.build();
}
```

After input objects are specified, we need to go back to the `createExecuteRobotMessage` and add the input as follows.

```
private Kcp.Message createExecuteRobotMessage() {
```

```
ExecuteRobot executeRobot = ExecuteRobot. newBuilder()
.setRobotUrl( REPOSITORY + "?project=" + P ROJECT + "&robot=" + ROBOT )
.setExecutionId( executionId )
.setInput(createMyInputs())
.build();
return Kcp.Message. newBuilder()
.setExecuteRobot(executeRobot)
.build();
}
```

Now you can send execute messages that start robot runs. The next step is to retrieve the robot results.

Step 6: Receiving robot results

A robot run can return `RobotResults`, `RobotRunStatus` and `RobotEvent` messages during its execution. To receive messages we need to set up a consumer on the result queue. The consumer picks messages off the queue as they arrive, and delegate further work. The queue is named the same way as the execute queue.

```
[NAMESPACE].KCP.[CLUSTER_NAME].Result
```

For example, `Kapow.KCP.Production.Result`.

In this example the consumer runs in a separate thread and keeps consuming until `stopConsumer()` is called. This specific consumer consumes all messages on the result queue. You can use this if you do not need to hand the results back to the specific executor. In KCP Tutorial 2: Consume Specific Results we set up an execution ID for specific consumer.

In the following code we set up a connection and a consumer for the result queue.

```
public void run() {
try {
//Create a ConnectionFactory
ActiveMQConnectionFactory connectionFactory = new
ActiveMQConnectionFactory( BROKER_URI );
//Create a Connection
Connection connection = connectionFactory.createConnection();
connection.start();
connection.setExceptionListener( this );
//Create a Session
Session session = connection.createSession( false , Session. AUTO_ACKNOWLEDGE);
//Create the destination
Destination destination = session.createQueue( RESULT_QUEUE );
//Create a MessageConsumer from the Session to the Topic or Queue
MessageConsumer consumer = session.createConsumer(destination);
...
```

Then we add the main consume loop where we consume the messages and parse them depending on the type of a message.

```
...
while ( consume) {
// Wait for a message for 1 second
Message message = consumer.receive( 1000 );
if (message instanceof BytesMessage) {
BytesMessage m = (BytesMessage) message;
byte [] bytes = new byte [( int ) m.getBodyLength()];
m.readBytes(bytes);
Kcp.Message kcpMessage = Kcp.Message. parseFrom(bytes);
System. out.print( "from sender: " + kcpMessage.getSenderId() + ": " );
switch (kcpMessage.getKindCase()){
case ROBOT_EVENT:
System. out.println( "RobotEvent: " + kcpMessage.getRobotEvent().getType().name());
break ;
```

```
case ROBOT_RESULT:
handleResult(kcpMessage.getRobotResult());
break ;
case SERVER_MESSAGE:
System. out.println( "Server Message: " +kcpMessage.getServerMessage().getMessage());
break ;
case ROBOT_RUN_STATUS:
System. out.println( "RobotRunStatus Message: returned objects: "
+kcpMessage.getRobotRunStatus().getLatestResultIndex());
break ;
default :
System. out.println( "unknown Message: " +kcpMessage.getKindCase().name());
}
}
}
...
```

The final handling of the returned data is simply unpacking the KCP object. In this example we just print the result to the output stream.

```
/**
* prints out a given result
* @param result
*/
private void handleResult(Kcp.RobotResult result){
Kcp.Structure output =result.getOutput();
System. out.println( "Result: object type: " + output.getTypeName() + " index: " +
 result.getIndex());
for (String key: output.getElements().keySet()) {
Kcp.Value value = output.getElements().get(key);
System. out.print( " \t " +value);
}
}
```

All the above examples together give you a code that can build an input object, execute a robot, and get returned events and results from the robot.

KCP Tutorial 2: Consume Specific Results

This is a modification of KCP Tutorial 1: Compile KCP, Connect to JMS Broker, and Send a Message and shows a different approach to consumption of messages. The source code can be found in `Tutorial2.java`.

If you want to extract messages related to your execution, instead of a global consumer use message selectors. Each message sent to the result queue has a message property with an execution ID. You can setup a consumer for a specific execution ID with a message selector like the following.

```
session.createConsumer(destination, "executionId='"+_executionId+"'");
```

Note More complex selectors can be created using the SQL92 condition format.

```
...
// Create the destination
Destination destination = session.createQueue( RESULT_QUEUE );
// Create a MessageConsumer from the Session to the Topic or Queue
MessageConsumer consumer = session.createConsumer(destination, "executionId = '" +
 _executionId + "'" );
while ( consume) {
...
```

With the message selector, only messages related to the specific `executionId` are handled by the consumer.

In KCP Tutorial 1: Compile KCP, Connect to JMS Broker, and Send a Message we set up a global consumer. Now we move the initialization of the consumer into the run method of the producer and pass the `executionId` to the consumer as follows.

```
...
public static class TutorialProducer implements Runnable {
...
public void run() {
//Start the consumer,
Consumer consumer = new Consumer( executionId );
thread(consumer, "Consumer thread" );
...
```

Finally we stop the consumer when we receive a `RobotEvent.ENDED` message.

```
...
switch (kcpMessage.getKindCase()){
case ROBOT_EVENT:
System. out.println( "RobotEvent: " + kcpMessage.getRobotEvent().getType().name());
if (kcpMessage.getRobotEvent().getType() == Kcp.RobotEvent.Type. ENDED) {
stopConsumer();
}
break ;
...
```

Running the code from KCP Tutorial 2: Consume Specific Results starts a producer that creates a consumer for your run. The consumer consumes all messages related to your run and closes down when the robot stops executing.

KCP Tutorial 3: Stop Robot Execution

Stopping a robot is a little different compared to the JMS communication in Tutorial 1 and Tutorial 2, because the `StopRobot` message is broadcasted to all RoboServers over a JMS Topic.

To connect to a topic is similar to connecting to a queue. The only difference is that we call the `session.createTopic(name)` instead of `session.createQueue()` as in the following example.

```
public void run() {
try {
// Create a ConnectionFactory
ActiveMQConnectionFactory connectionFactory = new
 ActiveMQConnectionFactory( BROKER_URI );
// Create a Connection
Connection connection = connectionFactory.createConnection();
connection.start();
// Create a Session
Session session = connection.createSession( false , Session. AUTO_ACKNOWLEDGE);
// Create the destination
Destination destination = session.createTopic( TOPIC );
// Create a MessageProducer from the Session to the Topic or Queue
MessageProducer producer = session.createProducer(destination);
producer.setDeliveryMode(DeliveryMode. PERSISTENT);
// Create a messages
Kcp.Message kcpMessage = createStopRobotMessage();
...
```

The rest of the connection setup is the same as for execute message.

When building a Stop Message all we need is the `executionId` as follows.

```
private Kcp.Message createStopRobotMessage() {
Kcp.StopRobot stopRobot = Kcp.StopRobot.
 newBuilder().setExecutionId( executionId ).build();
return Kcp.Message. newBuilder()
.setStopRobot(stopRobot)
.build();
}
```

When a RoboServer receives a Stop message, it stops the robot after the current step is executed and sends back a `RobotEvent` over the Result queue.